September 1990

UILU-ENG-90-2247
CRHC-90-13

Center for Reliable and High-Performance Computing

A STUDY OF SOME DESIGN CHOICES FOR IMPROVING THE PERFORMANCE OF A SHARED CACHE SYSTEM

John Wia Cheong Fu

Coordinated Science Laboratory
College of Engineering
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-90-2247 (CRHC-90-13)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research
6c. ADDRESS (City, State, and ZIP Code) 1101 W. Springfield Ave. Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code) 800 N. Quincy St. Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Joint Services Electronics Program 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00014-90-J-1270
8c. ADDRESS (City, State, and ZIP Code) 800 N. Quincy St. Arlington, VA 22217	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
A STUDY OF SOME DESIGN CHOICES FOR IMPROVING THE PERFORMANCE OF A SHARED CACHE SYSTEM

12. PERSONAL AUTHOR(S)
FU, John Wia Cheong

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	1990, Sept. 24	104

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	processors, cache cycle, multi-processor, simulation

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Processors with private caches have usually been assumed in the study of multi-processor systems, particularly when the number of processors is large. A private cache organization allows a processor to access only its own cache and some mechanism is required to maintain data coherence between all the caches. A number of solutions for the coherence problem have been proposed but all impose some degree of performance penalty on the system.

For a medium size multi-processor, e.g. where the number of processors is less than or equal to 16, an alternative scheme is to share a single cache among all the processors. A shared cache organization does not generally require a coherence mechanism and potentially offers better performance for accessing shared data. However, when multiple processors attempt to access the cache within the same cache cycle, conflicts occur such that one or more processors must wait for cache service. Furthermore, due to multiple reference streams, how a shared cache handles cache misses can have an important effect on system performance. ->

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

This thesis discusses the organization of a shared cache system and looks at the effect on its performance when the number of cache banks is increased and when buffer queues are added. In particular, by evaluating shared cache access conflicts and cache miss effects, we look at how these changes affect multi-processor performance.

To evaluate the effectiveness of increasing the number of cache banks and buffer queues on the performance of the shared cache system we simulate with a multi-processor model based on the Alliant FX/8. The model is driven by a set of address traces generated by a detailed simulation of an Alliant FX/8.

A STUDY OF SOME DESIGN CHOICES FOR IMPROVING THE PERFORMANCE OF A SHARED CACHE SYSTEM

BY

JOHN WIA CHEONG FU

BSc(Hons), University of Manchester, 1977

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1990

Urbana, Illinois

A STUDY OF SOME DESIGN CHOICES FOR IMPROVING THE PERFORMANCE OF A SHARED CACHE SYSTEM

John Wia Cheong Fu, MS
Department of Computer Science
University of Illinois at Urbana-Champaign, 1990
Edward S. Davidson, Advisor

Processors with private caches have usually been assumed in the study of multi-processor systems, particularly when the number of processors is large. A private cache organization allows a processor to access only its own cache and some mechanism is required to maintain data coherence between all the caches. A number of solutions for the coherence problem have been proposed but all impose some degree of performance penalty on the system.

For a medium size multi-processor, e.g. where the number of processors is less than or equal to 16, an alternative scheme is to share a single cache among all the processors. A shared cache organization does not generally require a coherence mechanism and potentially offers better performance for accessing shared data. However, when multiple processors attempt to access the cache within the same cache cycle, conflicts occur such that one or more processors must wait for cache service. Furthermore, due to multiple reference streams, how a shared cache handles cache misses can have an important effect on system performance.

This thesis discusses the organization of the shared cache system and looks at the effect on its performance when the number of cache banks is increased and when buffer queues are added. In particular, by evaluating shared cache access conflicts and cache miss effects, we look at how these changes affect multi-processor performance.

To evaluate the effectiveness of increasing the number of cache banks and buffer queues on the performance of the shared cache system we simulate with a multi-processor model based on

the Alliant FX/8. The model is driven by a set of address traces generated by a detailed simulation of an Alliant FX/8.

ACKNOWLEDGEMENTS

I would like to thank Professor Edward Davidson for his help and advice. The quality of writing in this thesis is due mainly to his careful reading. Our all too short association was both marvelously enlightening and enjoyable. I would also like to thank Professor Janak Patel for providing the support in Professor Davidson's absence. Without their patience during my struggles with some very diffcult times I would not have continued in Graduate school or deposited this thesis.

This work was completed while I was at CSRD, and I would like to thank all my friends there for all their help, particularly Hoichi Cheong and Nancy Warter. Additionally, I like to thank the professional staff at CSRD for the provision of some excellent working conditions.

During much of this work I was generously supported by Digital Equipment Corporation through the Graduate Engineering Education Program managed by Shirley Stahl. Finally, I would like to acknowledge the support from the management of the Advanced Vax Development group at DEC in particular Steve Jenkins and David Sager.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE PAGE

LIST OF TABLES

TABLE PAGE

CHAPTER 1

INTRODUCTION

Processors with private caches have usually been assumed in the study of multi-processor systems, particularly when the number of processors is large. A private cache organization allows a processor to access only its own cache and some mechanism is required to maintain data coherence between all the caches. A number of solutions for the coherence problem have been proposed but all impose some degree of performance penalty on the system.

For a medium size multi-processor, e.g. where the number of processors is less than or equal to 16, an alternative scheme is to share a single cache among all the processors. A shared cache organization does not generally require a coherence mechanism and potentially offers better performance for accessing shared data. However, when multiple processors attempt to access the cache within the same cache cycle, conflicts occur such that one or more processors must wait for cache service. Furthermore, due to multiple reference streams, how a shared cache handles cache misses can have an important effect on system performance. The reduction of shared cache access conflicts and the minimizing of cache misses and their effects on multi-processor performance is the subject of this thesis.

1.1. Thesis Outline

This thesis consists of five chapters. In this chapter the shared cache multi-processor organization and approaches to improve its performance are introduced. In Chapter 2 the multi-processor system is discussed in more detail. Specifically we discuss the various design choices

that can be made to increase its pe. ormance, such as buffer queues, number of cache banks and interleave schemes. We describe a request combining scheme to reduce the number of memory requests and a modified write back scheme to handle cache and memory updates.

In Chapter 3, the simulation model based on the ideas introduced in the previous chapters and the address traces used for experimentation are briefly described. Chapter 4 presents a set of simulation results from experiments with the multi-processor model. Chapter 5 concludes the thesis.

In this thesis we make a distinction between multi-processing and parallel processing. Multi-processing executes multiple processes on a multi-processor, while parallel processing executes a single process in a parallel form on a multi-processor. This distinction is important as the simulation address traces represent parallel loops. Thus, in this thesis, we are concerned exclusively with parallel processing on a shared cache multi-processor.

1.2. Cache organizations for multi-processors

There are two general ways to organize caches in a multi-processor; private caches and sh ared caches. Consider a system with p processors.

In a private cache organization there are p caches. Each processor, P_i, where $0 \leq i < p$, can access only cache C_i, and each C_i is connected to the main memory through an interconnection network. Typically all C_i's have the same capacity, organization and policies. Each P_i has conflict free access to C_i. If P_i causes a cache miss, a request is made to the memory through the network to load C_i with the missed data.

A well known problem with private cache organizations is that of maintaining data coherence among the caches. Consider a block of data b, where a block is the unit of data uniquely identified by a cache tag and is generally the unit of data transferred between the memory and

cache as the result of a cache miss. Assume there are valid copies of b in C_i as b_i and in C_j as b_j, where $i \neq j$. If processor P_i writes to b_i, then b_j must either be invalidated or updated such that when P_j reads from block b_j, $b_j = b_i$. How this is achieved is the responsibility of the coherence mechanism. The multi-cache coherence problem has been the subject of many studies. Generally all the many proposed solutions impose various penalties on the system, requiring combinations of network bandwidth, cache bandwidth [Tang76,CeFe78,ArBa84,PaPa84] and compiler support [Lee87,ChVe88].

The existence of multiple copies of blocks among the private caches can lead to program performance penalties. Consider again the two processors P_i and P_j, alternately reading and writing to a shared block b in a parallel loop. If P_i initially reads the block, the most up to date copy must be loaded into C_i as b_i. When P_i writes to the block, a typical coherence mechanism causes b_j, if it exists, to be invalidated. This invalidation insures that when P_j references b_j, a cache miss occurs and the up to date copy of the block is loaded into C_j from the memory or from another cache. When P_j writes into that block, the copy of the block in C_i is invalidated. The effect of consecutive read-write reference pairs by alternate processors is to "ping pong" the block between the associated caches. Each read reference to the block causes a cache miss, causing the block to be re-loaded from memory or from another cache, and each write invalidates the copy in the other cache. Such behavior can severely degrade parallel program performance with shared read-write data in a private cache system.

One possible approach to avoid potentially "ping ponging" shared read-write blocks between multiple caches is to make these shared blocks be non-cacheable. However, this imposes a performance penalty since all accesses to these blocks must reference the memory.

1.3. Shared cache systems

In a multi-processor system with a shared cache, all the processors are connected to the same cache. The advantages of this system include the following:

(1) Cache memories are expensive, so the sharing of the resource is desirable.

(2) As there is only one cache, there is no multi-cache coherence problem.

(3) The access of shared data and synchronization variables in the execution of parallel programs is critical for performance. In a private cache organization, these are the types of accesses most complicated by the coherence requirement, often resulting in these variables being non-cacheable. These problems do not exist for a shared cache organization. With a single cache, only a single copy of the shared variable can exist in the cache and it is available to all processors.

The disadvantage of the shared cache organization is that it must support multiple reference streams. First, as multiple reference streams must be serviced, the shared cache must support a higher access bandwidth than a single private cache. Secondly, some mechanism is necessary to connect the multiple processors to the shared cache and to resolve the priority of references. Sharing will generally make the cache access latency longer than for a private cache system. Thirdly, when more than one processor makes a reference to the shared cache in the same cache cycle, access conflicts may occur such that at least one processor must wait for at least one cache cycle for cache service. Due to these conflicts the cache access latency varies even for cache hit references. For a large number of processors such access conflicts become unacceptable. Therefore, shared cache organizations have been suitable only for a moderate number of processors, e.g. no more than sixteen. Finally, since multiple streams of references are being issued to the cache, how the shared cache handles cache misses has much more significance than in a private

cache system. If the shared cache is busy during the handling of cache misses, access conflicts greatly increase.

Shared cache organizations have not received as much attention as private cache organizations. The only extensive study known is [Yeh81]. In this study, the model used was a set of multi-stream pipeline processors connected to a shared cache and a high performance $L-M$ memory system.[1] This study presented an analysis of shared cache systems and some simulation results comparing their performance with private cache organizations.

Shared cache forms an integral part of the memory hierarchy for a set of machines manufactured by Alliant Computer Systems Corporation. The Alliant FX/8, [Alli85], is a system with eight Computation Elements (CE) connected through a crossbar to a four module shared cache, which in turn is connected to memory through a bus shared with Interactive Processors (IP).[2] The Alliant FX/8 forms a processor "cluster" of Cedar [KDLS86], a hierarchical multiprocessor being constructed at CSRD. The base system for this study is similar to the Alliant FX/8 organization.

1.4. Shared cache organization

In this study we look at some considerations in the design of a shared cache system for multi-processors. The general organization is shown in Figure 1.1, where p processors are connected through a cross-bar switch to a shared cache with c banks, which are further connected to m memory modules through a shared bus. Each processor is a pipeline processor which can issue overlapped references independently of one another. The design space of interest is that of

[1] This is a 2-dimensional organization of memory modules, with L busses and M cache modules per bus [BrDa77].

[2] There are two types of processors in the system. The eight CE's are each complete floating point processors with vector facilities and can be used in concert to execute programs in parallel. The IPs have no vector unit and are generally used for non-numerical programs, for instance the operating system and input-output processing.

medium size multi-processors where $p,c,m \leq 16$.

Given k processors, where $k \leq p$, accessing a particular bank within one cache cycle, access conflict exists if:

(1) $k>1$, or

(2) $k>0$ and the cache bank is busy.

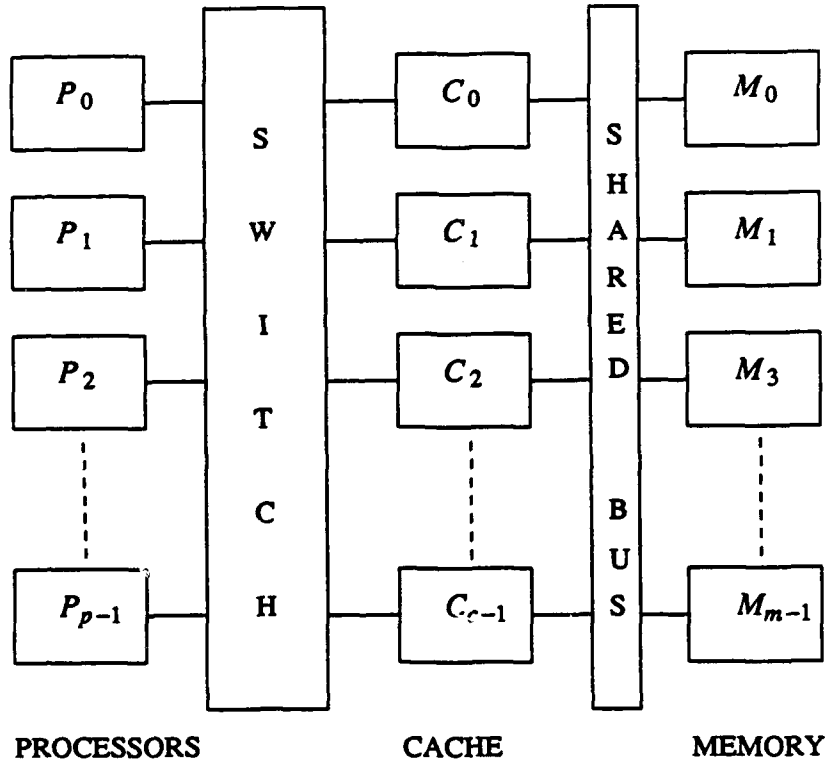


Figure 1.1: Multi-processor with shared cache

The first condition is due to multiple references to a shared resource. We call this a *multi-access* or *switch conflict*. The second condition is due to the service action of the cache and we call this a *bank-busy conflict*. Bank-busy conflicts occur when the cache cannot service further references due to previous cache misses. For instance, the cache may be handling a pending memory request or it may be busy being updated updated following a miss. If either conflict condition is true, then at least one processor must wait for cache service for at least one cache cycle. A reference that does not gain the cache, due to a bank-busy conflict, is held in some buffer, until the cache is available. If this buffer is already full then the processor must be stalled from issuing further references. Subsequent references to the same cache bank, from other processors, will compete with this held reference for the cache bank in the next cycle. Therefore an increase in the number of cache misses increases the probability of multi-access conflicts. In the following text, unless explicitly noted, conflict refers to both types. Some methods to reduce these conflicts and their effect on the processors are introduced in the following subsections.

1.4.1. Reducing multi-access conflicts

If the shared cache is organized as a monolithic structure, the probability of multi-access conflicts is very high. Such conflicts reduce the performance advantages of using multiple processors. A well known technique used to support parallel access to memory, is to organize the memory as a set of independent modules. The memory addresses are then distributed, or interleaved, amongst the modules in some ordered manner. Multiple banks and address interleaving can also be used to organize a shared cache. In our simulations we look at the effect of varying the number of cache banks and various interleaving schemes on conflicts. We also show how reducing the number of multi-access conflicts does not necessarily contribute to an increase in system performance for some address traces.

1.4.2. Minimizing the effect of conflicts

Increasing the number of cache banks only provides the potential for parallel access. Multi-access conflicts still occur if more than one processor attempts to access the same cache bank in the same cycle. The result of these conflicts can potentially lead to processor pipeline stalls. Using delays to improve the schedule performance of a pipeline was suggested by [PaDa76]. A delay is defined as a pipeline segment that performs no data transformation but holds a task state for one pipeline cycle. A set of n consecutive delay segments take n cycles to traverse, assuming single cycle delay segments.

A more flexible structure is a queue, which is similar to a set of consecutive delay segments except that the number of cycles taken for a new request to arrive at the head of the queue is equal to one plus the number of requests already in the queue when the new request arrived, assuming that one request is removed every cycle. For instance, a n entry queue with m stored requests, where $m \leq n$, takes at least $m+1$ cycles to traverse. Figure 1.2 shows a buffer queue with 3-stages, B_0, B_1 and B_2, connecting two latches L_0 and L_1. Assume that a stream of requests arrives as input to L_0 on consecutive cycles, where the buffer queue is initially empty. A request is moved from L_0 into B_0 after one clock period. In the next cycle, if L_1 can accept a request

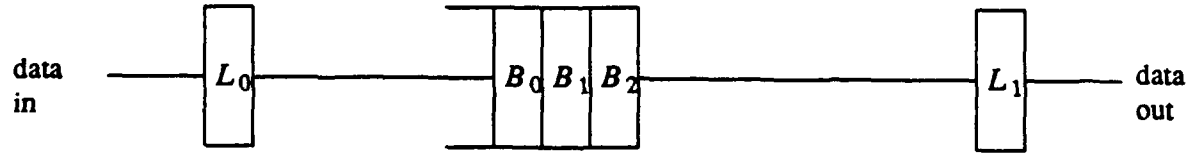


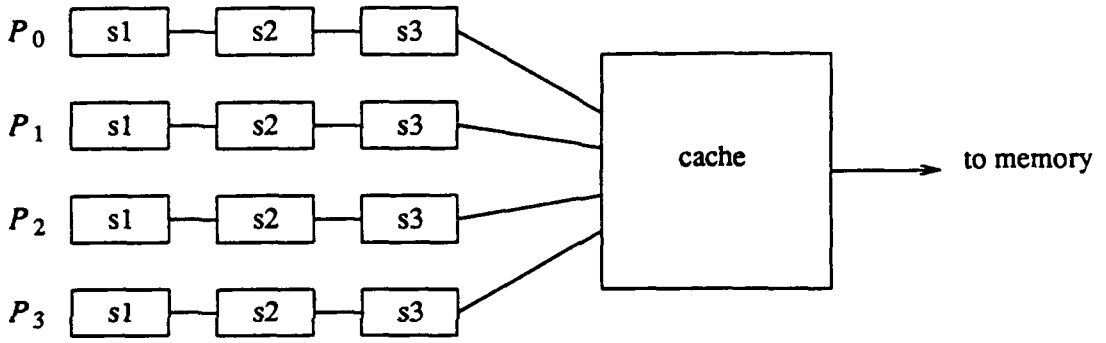
Figure 1.2: Buffer queue

then the contents in B_0 is moved into L_1. Otherwise, the request is moved into B_1. In either case, the next input request is moved from L_0 into B_0. If L_1 becomes available on the next cycle, the contents of B_1 is moved to L_1, B_0 is moved to B_1 and L_0 to B_0. The queue buffers the issuing of data by L_0 from the performance of L_1 for up to three requests. However, if L_1 is always able to accept data, there is only a one cycle latency for moving data from B_0 to L_1.

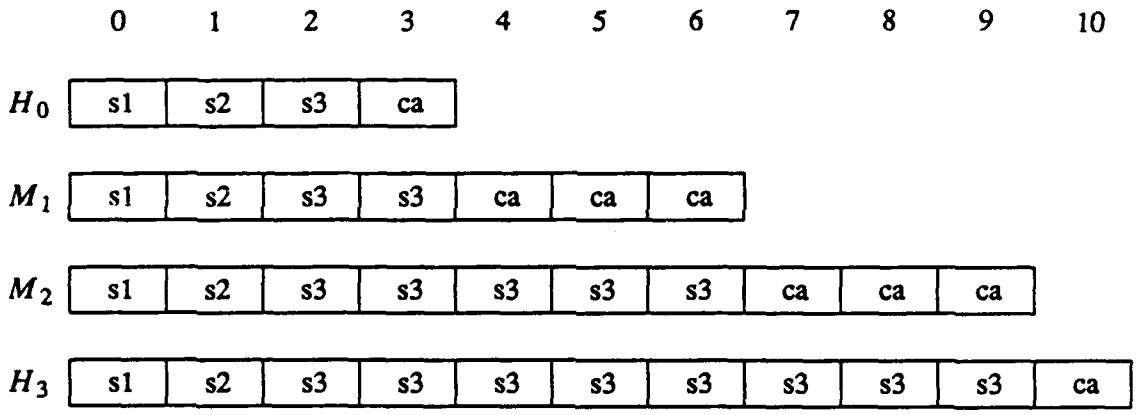
The use of queues or buffers to smooth out operand access delays has been implemented in a number of machines including IBM 360/91 [AnST67], ZS-1 [Smit87] and ICL2900 [Buck78] and is a basic mechanism in the DEA [Smit84] and SMA [PlDa83] architectures. We examine the use of queues to minimize the effect of multi-access conflicts at the cache.

In a cache that serves multiple reference streams, cache misses can have a severe performance effect on subsequent references. Consider the organization in Figure 1.3a where four pipeline processors (p_0, p_1 p_2 and p_3) share a cache. Each processor pipeline has 3 segments, s1, s2 and s3, with each segment taking one cycle to complete its function. References issued by each processor pass through that processor's segments to address the cache. A reference takes one cycle to complete in the cache if it is a cache hit. For a miss, two additional cycles are necessary to get the data from the memory. (A two cycle memory access time is selected to keep the example simple.) Consider the total time spent processing and serving a reference, including the instruction passing through the processor pipeline prior to the reference arriving at the cache. Assume that a reference remains in S3 until the arbitration mechanism grants it exclusive access to the (single bank) cache. A hit reference will thus take four cycles (plus conflict cycles if any) to complete, and a miss will take six plus conflict cycles.

Figure 1.3b and 1.4b show timing diagrams for the progress of four references H_0, M_1, M_2, H_3 issued by processors p_0, p_1, p_2 and p_3 respectively where H_0 and H_3 will hit in the



a) Four pipeline processors with a shared cache



b) Reference timing diagram

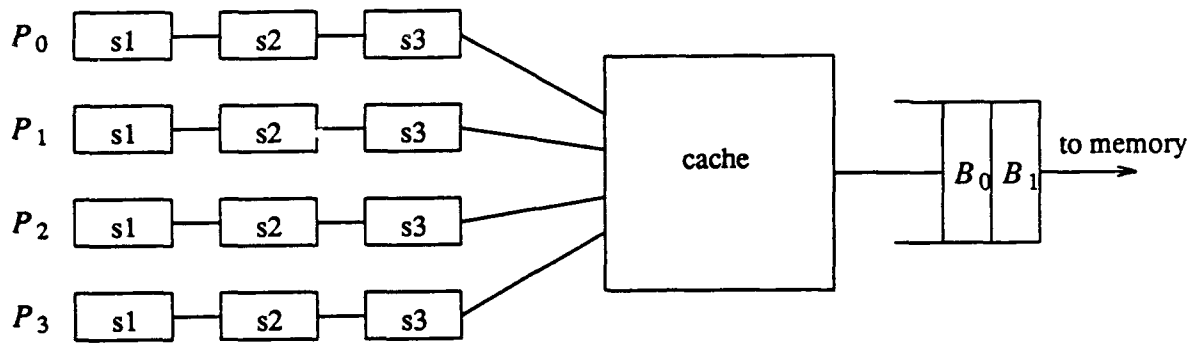
Figure 1.3: Shared cache without buffer queue

cache while M_1 and M_2 will miss. Assume p_0 has the highest priority and p_3 the lowest and the system is initially idle. In Figure 1.3b, due to the lack of a buffer, the cache can only handle one cache request, hit or miss, at a time. Reference H_0 has the highest priority and is able to complete in four cycles. The other references are stalled in s3 of their respective processors while H_0 address the cache. The stall is shown as the additional s3 cycles. Reference M_1 gets the cache

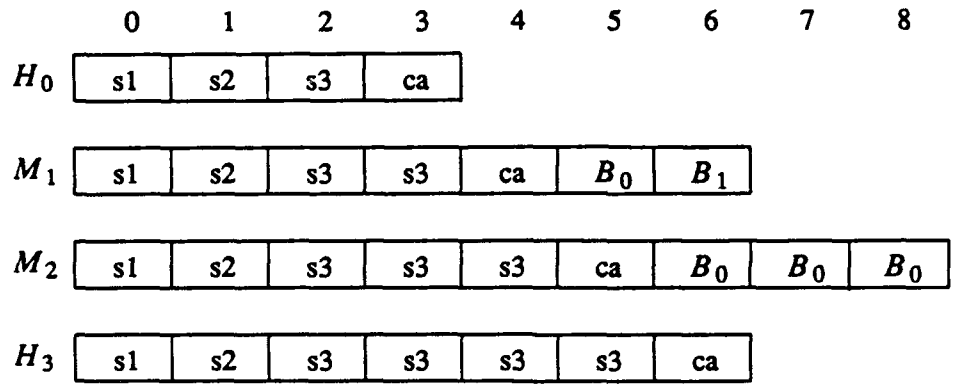
when H_0 completes. As this reference is a miss it must spend one cycle in the cache to find out that it was a miss and two additional cycles to get its data from the memory. Holding M_1 in the cache while it is waiting for memory data causes further stalling of references M_2 and H_3 in s3 of processors p_2 and p_3, respectively. When reference M_1 completes, M_2 addresses the cache and as it is also a miss, it causes further stalling of reference H_3. Reference H_3 gains the cache after reference M_2's data has been fetched from memory. H_3 completes after one cycle as it is a cache hit. As the cache can only handle one request at a time, the cache access latency for a particular reference, whether it is issued by the same processor or by a different processor, is highly dependent on the service of the immediately previous cache accesses.

In Figure 1.4a, a 2 entry buffer queue is located after the cache. When reference M_1 misses it is moved out of the cache and forward into B_0 of the queue while making a request to the memory system. This releases the cache, in the next cycle, for M_2. As M_2 is also a miss, it is moved into B_0, forcing M_1 to be moved forward into B_1. Queuing these miss requests enables reference H_3 to address the cache in the next cycle. As H_3 is a hit, the reference is completed in the cache in one cycle. By pipelining the cache misses forward into the buffers, the resource conflict between references is weakened. This increased independence potentially decreases the average number of cycles to complete the references. A miss address is held in a buffer until the memory access has been completed and the requested is block returned to the cache. The block is written into the cache using the buffered address and, for a read request, the desired data is sent to the processor. For simplicity we have assumed that updating the cache and sending the data to the processor does not cause any busy conflicts with incoming requests.

By organizing the buffers in the form of a queue, the latency for completing references is reduced. In the simulation model we use buffer queues both before and after the cache to reduce



a) Cache with 2-entry buffer queue



b) Reference timing diagram

Figure 1.4: Shared cache with buffer queue

the effect of crossbar switch conflicts, as well as cache miss effects.

The handling of concurrent cache misses in the system presents some control issues. For example, a second miss to the same cache block may occur while the first miss is still being served by the memory. A second memory request must not be issued when this occurs. Further-

more, when miss references can potentially be served out of order, a mechanism is needed to resolve access hazards with minimal cost in performance. These control issues are discussed in Chapter 2.

1.5. Simulation model

To evaluate the effectiveness of increasing queue depths and the number of cache banks on the performance of the shared cache system, a simulation model, based on the Alliant FX/8, was written. The model is driven by a set of address traces generated by a detailed simulation of the Alliant FX/8. The simulation environment is described in Chapter 3, and the results of these simulations are presented in Chapter 4.

CHAPTER 2

IMPROVING THE PERFORMANCE OF A SHARED CACHE

In this Chapter we look at the multi-processor organization shown in Figure 1.1 in more detail. Specifically we develop the ideas, introduced in the previous Chapter, for the reduction and minimization of conflicts in the shared cache organization.

The first part of this Chapter presents the organization under study. The following sections discuss some design choices that can be made to improve performance. We look beyond the questions of cache mapping and replacement schemes and block sizes. Instead, we concentrate on the organization of reference buffers, overlapping of references, multiple cache banks and interleaving of addresses.

2.1. System architecture

The system of interest is shown in Figure 2.1, where a set of pipelined processors is connected to a set of cache banks through a crossbar switch and the cache banks are connected to the memory modules through a shared bus.

There are a number of ways to connect the processors to the cache banks and the cache banks to the memory modules. They range from the shared bus to the crossbar switch. In this study we restrict our attention to the common case of the crossbar switch for the processor to cache connection and the shared bus for the cache to memory connection. The processor to cache connection is expected to have a high bandwidth requirement. For a medium size multiprocessor, a crossbar is an appropriate choice. Conversely the cache to memory connection is

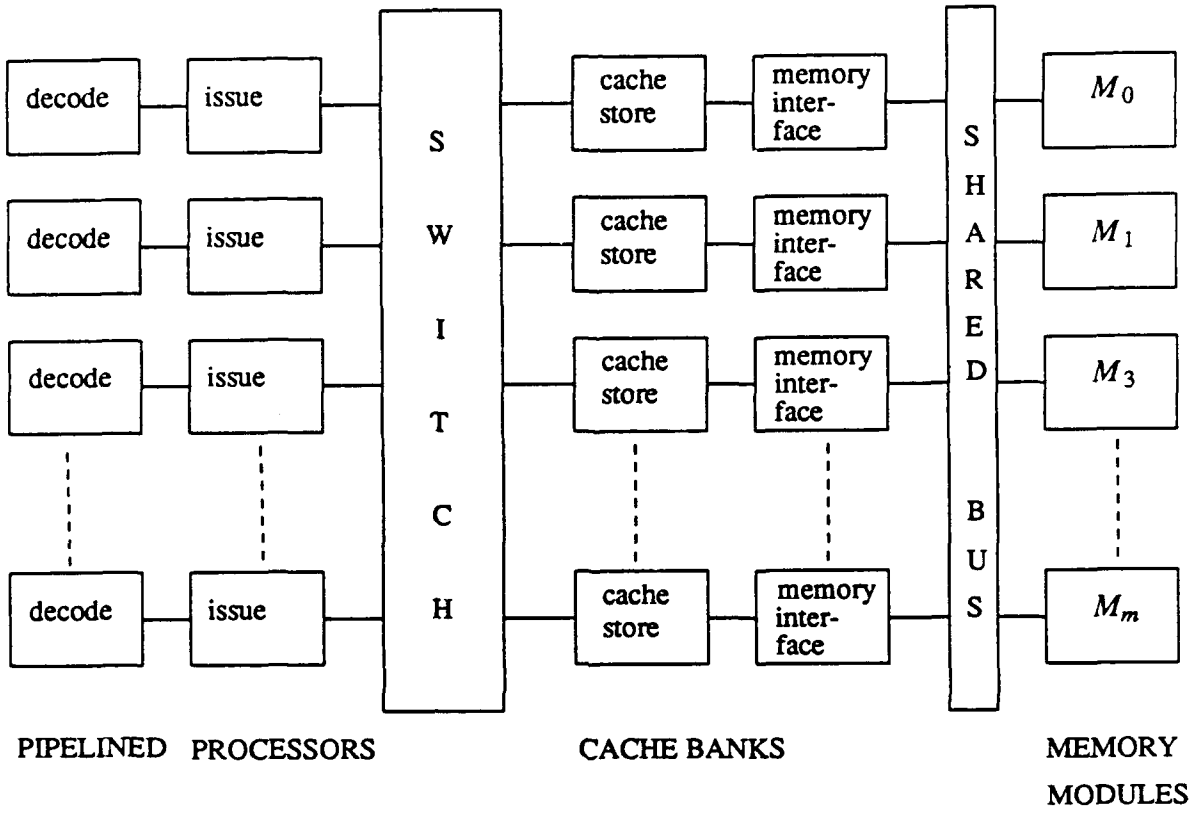


Figure 2.1: Multi-processor with shared cache

expected to have a relatively low bandwidth. Here, a shared bus is an appropriate choice.

When multiple processors need to access a shared cache, a monolithic organization of the cache is unlikely to satisfy the bandwidth requirements. A better approach is to organize the cache as a set of independent banks, each able to serve its own stream of references. Hence, while the cache is logically a single cache, it physically consists of multiple banks. Each bank has its own tag and data stores so that all banks can be accessed concurrently, and each bank has its own interface to the shared bus. The location of memory blocks in the cache depends on the mapping and address interleaving scheme used. As a function of the interleaving scheme chosen,

individual blocks can be contained within a single cache bank or spread across two or more banks. These issues are discussed in Section 2.4.

All of the memory modules operate independently from one another and each consists of a data store and the hardware necessary for address decoding, data access, bus arbitration and driving the bus for returning data to the cache banks.

In this thesis we are not primarily concerned with how data is returned from the cache banks to the processors. We assume that data can be sent to the processors while the cache banks are being updated following a cache miss and that some connection is available to broadcast data back to multiple processors.

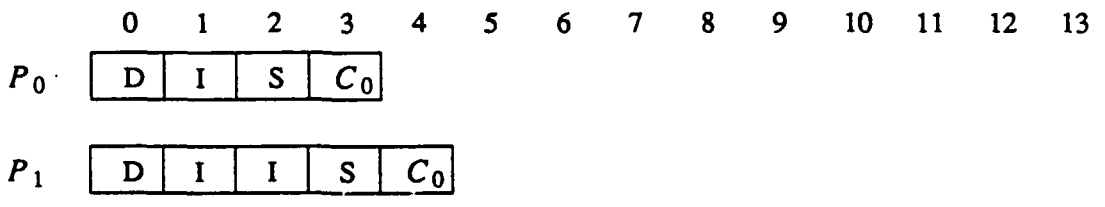
2.2. System performance degradations

Each of the pipeline processors consists of 2 stages or segments: a decode segment, which decodes the instructions, and an issue segment, which forms the address and issues references to the cache. As we are primarily concerned with data access we do not consider the other functional units of the processors. Each of the two segments takes one time unit (or pipeline cycle) to complete its function and can only operate on or hold one reference in any particular cycle.

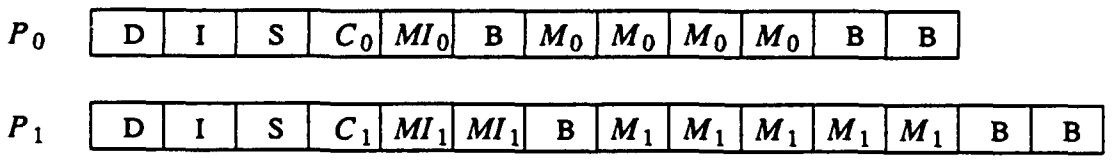
The processors are connected to a crossbar switch which routes cache references to the requested cache banks. The switch can route a reference to any cache bank in one cycle. Each cache bank is also considered to consist of two segments. The cache store segment contains the cache and tag store and appropriate hardware to determine if a reference is a cache hit or miss. The memory interface segment takes a cache miss and constructs the memory request, arbitrates for the bus, sends the memory request to the appropriate memory module and receives the returned memory data. For our discussions we assume that the cache store and memory interface

segments also take one cycle each to complete their functions and can only operate on or hold one cache reference in any particular cycle.

Consider two processors, P_0 and P_1, each issuing a reference to the same cache bank C_0 in the same cycle. If both hit in the cache then the timing is as shown in Figure 2.2a. Processor P_0 has the highest priority for the switch and therefore accesses cache bank C_0 first. Processor P_1 must hold its reference in the issue segment, for an additional cycle, due to the multi-access



a) Cache hit references; P_0 and P_1 reference the same cache bank C_0



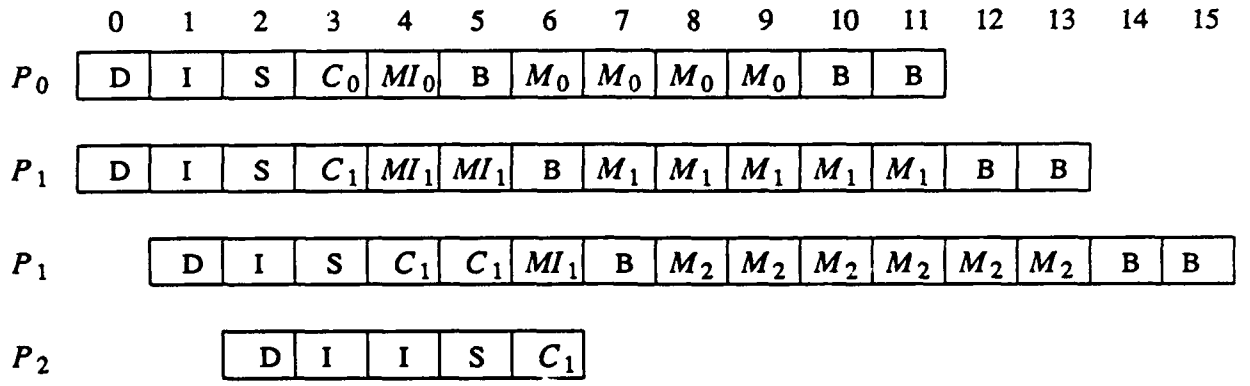
b) Cache miss references; P_0 and P_1 reference distinct cache banks C_0 and C_1 respectively

D: decode segment
I: issue segment
S: crossbar switch
C_n: cache bank n
MI_n: memory interface of cache bank n
B: shared bus
M_n: memory module n

Figure 2.2: Multi-access conflict

conflict, until the cache bank becomes available to service its request at time 4. If there are p processors and c cache banks, then the best performance per processor occurs when $p \leq c$ and in each cycle the processors all reference distinct cache banks and all references are cache hits. In this situation all processors can proceed in parallel at full speed, with no conflicts. If k processors (where $k \leq p$) reference the same cache bank in a given cycle and all references are cache hits, then the lowest priority processor must wait for $k-1$ cycles to reference the cache. Hence, the crossbar switch and the shared cache can be a major source of performance degradation, particularly if each processor has the capability to issue a reference in every cycle.

Consider again processors P_0 and P_1 where each processor issues a reference to a distinct cache bank in the same cycle. Figure 2.2b shows the timing for two cache misses where we assume that it takes 1 cycle to issue a request to memory on the shared bus, that the memory takes 4 cycles to access the data, that it takes 2 cycles to return the data (on the same bus) to the cache bank, that the interleaving scheme is such that no block is split across multiple cache banks or multiple memory modules and that there are no buffer queues. The case shown further assumes that the blocks for these two miss references are contained in distinct memory modules. As P_0 and P_1 reference distinct cache banks, they can access their respective caches concurrently. However, as both references are cache misses they must both issue requests to memory. MI_0 is assumed to have the the highest priority for the shared bus and sends P_0's request to memory first; P_1's memory request must be held in MI_1 until the bus becomes available. Similarly, as shown in Figure 2.2b, the return data for the second request must be held in memory bank M_1 for one extra cycle until the bus becomes available to it. Hence, the shared bus is sometimes a source of performance degradation when accessing memory as the result of cache misses.



P_0 references cache bank C_0, P_1 and P_2 reference cache bank C_1

Figure 2.3: Cache bank busy conflict due to cache miss

As stated in Chapter 1, in a shared cache system the performance of the memory interface can also degrade the whole system performance by increasing cache bank busy conflicts. Figure 2.3 shows processor P_0 and P_1 as in Figure 2.2b, except that processor P_1 also starts a second reference at time 1 to the same cache bank, C_1, as its previous reference, and it is also a cache miss. This block requested by the second reference made by P_1 is assumed to be located in memory module M_2. As P_1's first reference was stalled in the memory interface segment for one cycle, P_1's second miss must stall in the cache store segment for one cycle. A third processor, P_2, now starts a reference at time 2 to the same cache bank (C_1) as P_1's two references but this is a cache hit reference. As the second reference of P_1 was stalled in C_1 for one cycle, there is a bank busy conflict and the P_2 reference must stall in its issue segment as the cache is unavailable until time 6.

In the above examples we have looked at performance degradation in the shared cache system due to multi-access and bank busy conflicts. Each of these conflicts can degrade the performance of the system by causing references to stall in any or all of the segments of the processor pipeline, the cache bank, or its memory interface, and the memory module. Stalls in the switch and the bus are not permitted and it is assumed that the control logic has access to sufficient global status information to prevent such stalls and instead forces the stall to occur at the earlier segment when necessary. In the following sections we discuss the use of buffer queues to reduce the effect of these sources of performance degradation.

2.3. Buffering references to the cache

In the previous discussion it was shown that multiple references to the same cache bank can degrade performance by stalling processors keeping them from issuing references at their maximum rate. This degradation can be quite severe relative to the no miss, no conflict peak performance.

To minimize the effect of multi-access conflicts on processor performance a set of buffer queues can be located between the processors and the switch or between the switch and the cache banks. Figure 2.4 shows the two organizations of the buffer queues. To distinguish between these two sets of queues, the former will be referred to as *processor queues* and the latter as *cache queues*. Figure 2.4a shows the location of the processor queues. Each processor queue is associated with a particular processor, which places its references into the queue and stalls from issuing further references when this queue becomes full. The reference at the head of each processor queue, in a particular cycle, is the reference to be routed through the switch to its requested cache bank. A reference cannot be taken from the body of the queue even if its requested cache bank is available when the reference at the head of the queue is blocked. Hence,

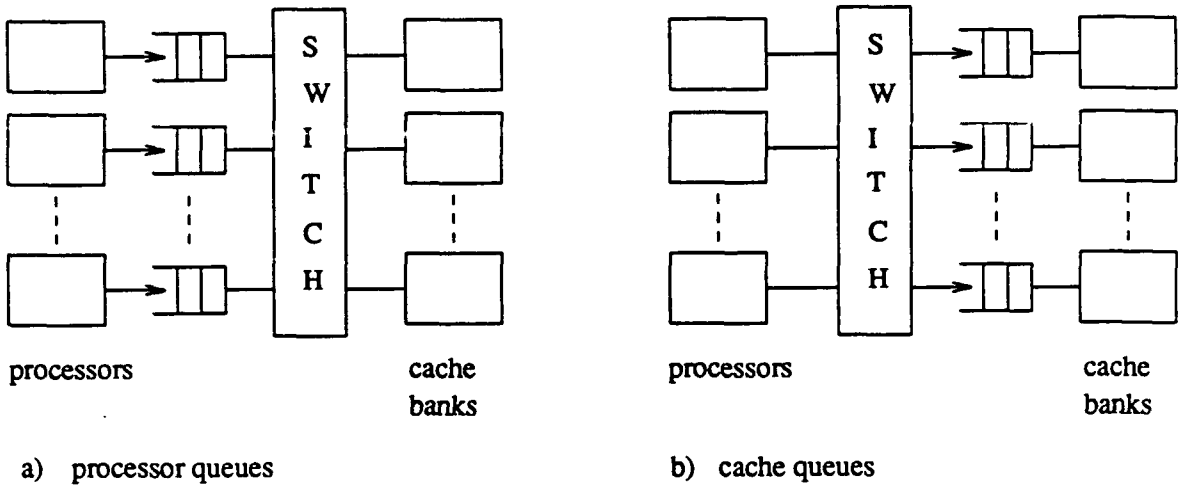


Figure 2.4: Buffer queue organizations

an issued reference can be blocked in the body of the queue by other references issued earlier by the same processor or at the head of the queue by references at the heads of other queues with a higher priority that are destined for the same cache bank.

Figure 2.4b shows an alternative scheme which associates the queues with the cache banks. In this organization each cache bank has a cache queue to hold incoming cache requests. The reference at the head of the cache queue is selected to address the cache bank, when it becomes available. The switch can route references through to a particular cache bank until its cache queue becomes full. Cache queues intuitively seem preferable to processor queues since the references are separately queued for each requested cache bank and thus the blocking problem with processor queues is avoided, i.e. with cache queues a reference within the queue can only be blocked by earlier references to the same cache bank.

In the previous examples of access conflicts (Figure 2.2 and Figure 2.3) we have assumed that each instruction processed in the processor pipeline makes a memory reference. The processor can also process instructions that do not make a memory reference such as an arithmetic instruction. Since the focus of this thesis is memory accessing, a non-memory instruction is shown as occupying only the D and I segments during its processing. For simplicity the completion of non-memory instructions beyond the D and I segments is assumed to have no effect on the processor pipeline under study and therefore the execution unit pipeline segments are not shown in the examples. In the next example we use a non-memory instruction to demonstrate the benefits of a processor queue. Figure 2.5 shows how a processor queue can reduce processor stalls when a multi-access conflict occurs. In Figure 2.5a the processors P_0 and P_1 make references to the same cache bank and all references are hits. Processor P_0 processes one memory referencing instruction and P_1 processes 3 memory references and 1 non-memory instruction. At time 0 both processors start their first instructions. As P_0 has the highest priority it accesses the cache first. Processor P_1 must stall its reference in its I segment for one cycle until the switch path to C_0 becomes available. Processor P_1 issues its 2nd memory reference at time 1 and this is stalled in the D segment for a cycle due to the one cycle stall of the 1st memory reference in I. Processor P_1 now attempts to issue a non-memory instruction and a 3rd memory reference. However, due to the multi-access conflict at time 2 between P_0 and P_1, the non-memory instruction and the 3rd reference cannot enter the processor pipeline until time 3 and 4, respectively and the 3rd reference cannot access the cache until time 7.

Figure 2.5b shows the effect of a single entry processor queue. With the addition of a processor queue, P_1's 1st memory reference is loaded into the processor queue when the multi-access conflict occurs. This allows the 2nd reference, the non-memory instruction and the 3rd reference to enter the processor pipeline in consecutive cycles. The 3rd reference then accesses

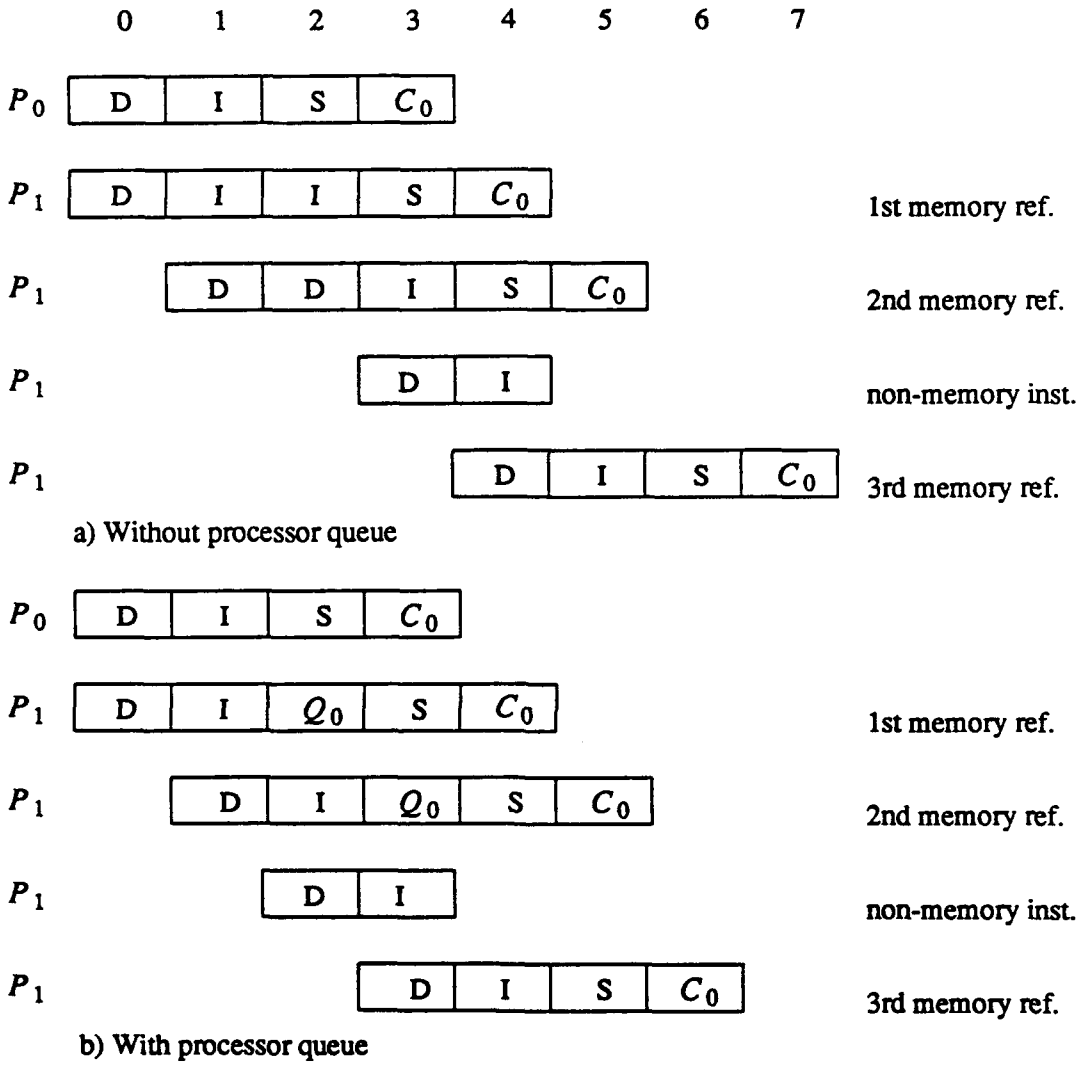


Figure 2.5: Effect of processor queues

the cache at time 6 one clock earlier than when no processor queue is present.

Figure 2.6 shows how cache queues located between the switch and the cache bank can reduce processor stalls during a cache bank busy conflict. In Figure 2.6a processor P_0 makes a cache miss reference to cache bank C_0, and the first reference from P_1 makes a cache hit refer-

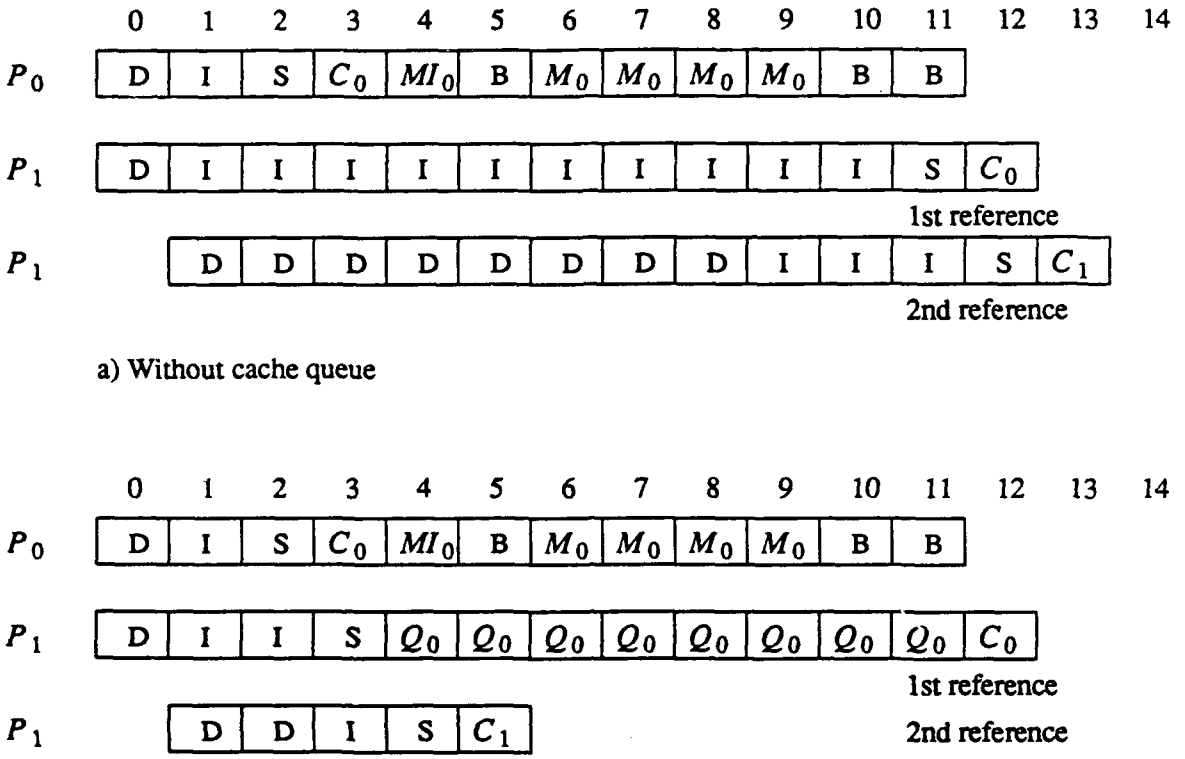


Figure 2.6: Effect of cache queues

ence to the same cache bank. If there are are no processor queues before the switch and the cache can only handle a single miss reference at a time then P_1's first reference is blocked from C_0 by a cache bank busy conflict until time 12 when the conflicting cache bank is completely updated. Note that during the time units 10 and 11, when the cache bank is being updated, the cache bank is unavailable to service any requests. If P_1 issues a second reference to another cache bank (C_1) then this reference is blocked from C_1 by the first reference until time 13. The bank busy conflict at C_0 stalls P_1 from issuing any further references until the first reference has

passed through the switch. Figure 2.6b shows how an organization with cache queues allows the first reference of P_1 to be moved from the issue stage of P_1 into the cache queue of C_0 allowing P_1's second reference to access C_1 immediately after the blocking reference has been moved into the cache queue.

In the previous examples we have discussed organizations with either a processor queue or a cache queue. An organization with both processor and cache queues is expected to be advantageous in some cases. The simulations we described in Chapter 3 use both processor and cache queues.

2.4. Cache bank organization

A cache bank consists of two segments: the cache store segment and the memory interface segment. The cache store itself consists of two sections, the data section and the tag section. Data loaded from the memory is held in the data section and information to identify its corresponding address is held in the tag section. The cache mapping scheme determines how the memory address is mapped into the cache address space, which parts of the address are held in the tag store, and how cache hits are determined. Various mapping schemes have been investigated as the subject of a number of studies [Smit82]. Since we are interested in the effect of queue buffering on processor performance, the selection of the cache mapping scheme is not of prime importance. We select the direct mapping scheme in which each memory address can only be mapped into one particular cache location. The direct mapping scheme is the simplest of the cache mapping schemes and generally provides the lowest hit ratios for a particular size of cache and hence the best stress test for the buffering system.

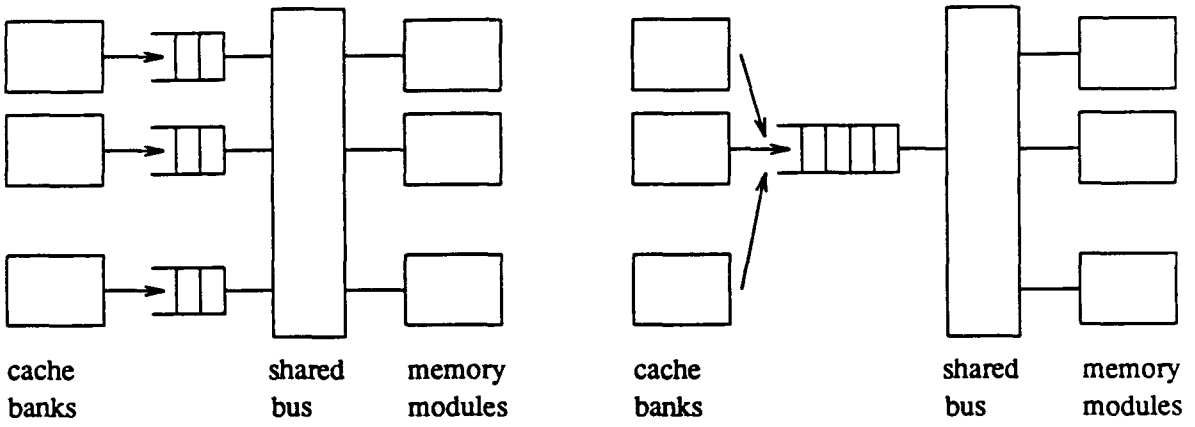
The memory interface segment of the cache is used by a reference only when it misses in the cache store. The segment constructs the memory request, arbitrates for the shared bus, sends

the memory request to the memory module and receives the data that is returned from the memory. Previous cache studies have tended to ignore the role of this part of the cache. As previously discussed, for an organization such as the shared cache, how a reference miss is handled has a significant direct effect on the performance of the cache as a whole. In the following section we discuss overlapping and queuing of cache misses.

2.4.1. Buffering cache misses

By organizing a cache bank's store and memory interface functions as two segments in a pipeline, two cache misses can be held in the cache bank before it becomes busy to further requests. However, in a shared cache, as the cache banks are shared by all the processors, as few as 3 miss references from different processors to the same cache bank will lead to a cache bank busy conflict even if each processor were limited to one pending reference. Stalling references in the cache bank is caused by insufficient buffering to hold cache misses while the memory requests are pending. Hence, as with reference buffering, we use a buffer queue, referred to as a *miss queue*, between the memory interface segment and the shared bus to hold miss requests while they are waiting for bus and memory service. The role of the miss queue is to decouple the performance of the shared bus from the cache banks and subsequently reduce the probability of cache bank busy conflicts.

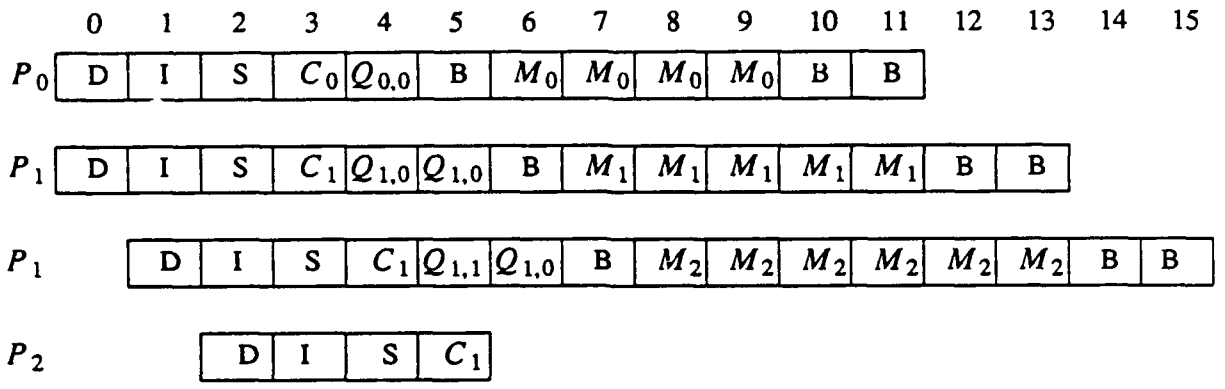
The miss queues can be organized as a common queue used by all cache banks or as separate queues for each bank, as shown in Figure 2.7. However, a single miss queue for all banks is inappropriate as it creates conflicts in loading the queue when misses occur simultaneously in several cache banks. Hence, in all subsequent discussions and in the simulations, we assume there is one miss queue per cache bank.



a) one miss queue per cache bank b) single miss queue

Figure 2.7: Organization of Miss queues

Figure 2.8 shows the effect of a 2-entry miss queue for the same scenario as shown in Figure 2.3. Note that with a cache miss queue the function performed in the MI_n segment is carried out concurrently with the initial loading of the miss request into the cache miss queue. With available buffering in the miss queue the first and second cache miss references of P_1 are not stalled either in the cache store or the memory interface segment. This subsequently frees the cache store segment for the reference from P_2 which completes its cache hit access in the minimum time. The miss queues have two effects on the system. First, they increase the number of misses held in a cache bank and consequently increase the possible number of memory requests simultaneously pending service by the memory modules. Second, they reduce the probability of cache bank conflicts by allowing cache misses to move out of the cache store segment, making it available to other references.



$Q_{m,n}$: load miss queue position n at cache bank m

Figure 2.8: Effect of 2-entry miss queue

The operation of the miss queues differ from that of the processor and cache queues. Both the processor and cache queues are simple first-in-first-out buffering structures. The miss queues differ in two respects. First, the miss queues must continue to hold a cache miss reference after its memory request has been sent to memory. Retaining this request information is necessary in order to identify the return data from memory, to write the data to the proper cache bank location, and to perform address matching. Only after the data has been written into the cache bank can the original miss reference be removed from the queue. Secondly, some mechanism is necessary to resolve read-write hazards. How address matching and read-write hazard resolution is achieved is discussed in the next 3 sub-sections.

Queues could also be located at the input ports of the memory modules. Such memory queues allow a busy memory module to continue to accept requests until its queue becomes full. Two performance benefits result from using memory queues. First, by using idle bus cycles to send miss requests to busy memory modules, contention on the shared bus is reduced. Second,

by allowing miss requests to busy modules to be issued across the bus, later queued miss requests to free modules can be issued and served earlier. The evaluation of memory queues is, however, beyond the scope of this thesis.

Since miss blocks returning from the memory are given the highest priority for accessing the bus and for updating the cache, memory output queues and queues for blocks to be loaded into the cache are not necessary.

2.4.2. Organization of the miss queues

In the following discussion we restrict ourselves to read miss references. In Section 2.3.3 we discuss how writes are handled in this system. Furthermore, throughout Section 2.3 we assume that a cache block is contained within a single cache bank. The complications that arise when a block is spread across two or more cache banks are discussed in section 2.4.

Buffering miss references allows multiple memory requests to be overlapped, i.e. requests can be made to the memory prior to the completion of previous requests. While this overlap can potentially achieve a higher throughput, complications arise which can reduce its effectiveness. Consider read references to words *i* and *j*, where both words fall within the same cache block. Suppose the word *i* reference misses in the cache and generates a fetch to the memory for the block. If some processor issues a read to word *j* while the word *i* request is still pending, then word *j* is also a miss. Since we have a pipeline and buffering, this second miss, unless detected and blocked, will be moved to the buffers and subsequently issue another memory read for the same block. This situation will occur frequently in our system because:

(1) processor references are to words while memory requests are for blocks, which contain several consecutive words,

(2) vector references with a stride less than the block size are not uncommon, and

(3) multiple streams of scalar and vector references are issued concurrently among the processors.

When multiple misses are simultaneously pending for the same block, it is undesirable to issue multiple memory requests for three reasons. First, these additional memory requests use up bus and memory bandwidth. Second, these memory references are redundant since the additional misses can all be satisfied by the first pending request, effectively reducing their memory access latencies. Third, some mechanism is necessary to maintain coherence of the requested blocks.

To prevent multiple fetches of the same block, any addresses that miss in the cache must be compared with any pending read miss addresses already in the miss queues before they are loaded into the miss queues. An address match means that the new read miss address can be satisfied when the pending block is returned. Hardware is required not only to perform this matching, but also to record the match status so that returning blocks will satisfy the pending requests of all the processors waiting on this block.

Satisfying the matched requests can be handled in one of two ways. In the first method, Figure 2.9, all matched addresses are placed into a separate "matched" queue. Only those miss addresses that actually generate a memory request to main memory are held in the miss queue. When a pending block is returned and written to the cache, all references held in the matched queue, address the cache again. Addresses that hit now return data to the processor and are removed from the matched queue. Those that miss again, are re-circulated back to the matched queue. This scheme is attractive as it is relatively inexpensive and simple to control. However, feeding back the matched addresses presents several performance issues. First, the matched

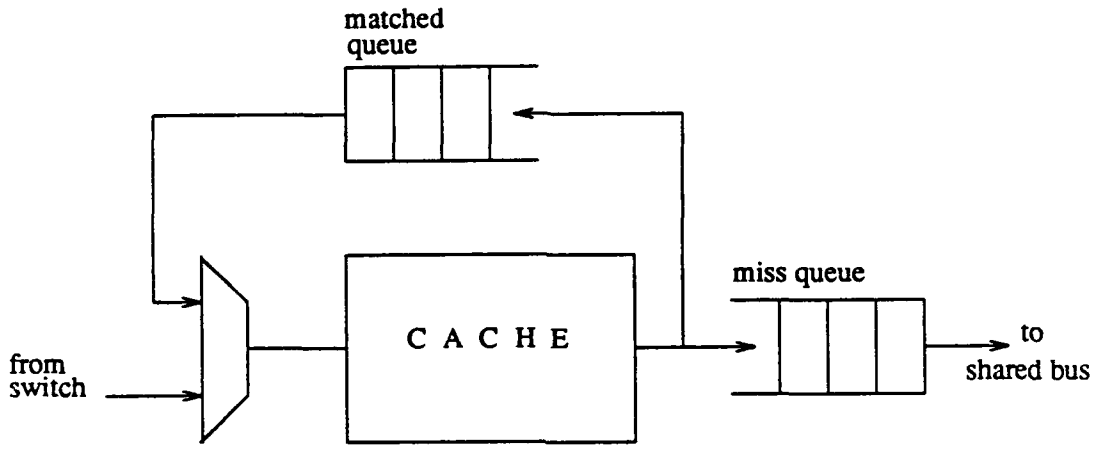


Figure 2.9: Separate matched reference queue

addresses must be serially fed-back to access the cache and all matched addresses must access the cache whenever a new block of data is written into the cache. Furthermore, to address the cache these matched addresses must use some of the cache access bandwidth, which can potentially lead to service conflicts with incoming requests. To lower the cache access contention effect of the feedback of the matched addresses, they could be assigned a lower priority than new requests from the switch. This priority, however, would delay the returning of missed data to due those requesting processors whose addresses are held in the matched queue and these processors might be stalled, resulting in another source of performance degradation.

An alternative scheme maintains all state information concerning the matched status of the miss references within the miss queues themselves. When a pending block returns from memory, all addresses that matched that particular block are removed from the queue, and the corresponding processors are sent their data which is extracted directly from the block returned

from memory. This scheme does not use any additional cache access bandwidth for serving matched requests, but the hardware required to maintain the match status is quite complicated and expensive. However, since it is expected to have better performance than the first scheme, this second scheme is used in our system.

To manage the matching of requests we use a mechanism based on [Toma76]. Figure 2.10 shows the general organization used in each cache bank. The miss queue consists of an associative store to hold the pending miss addresses, any corresponding data and a tag number. A separate associative store is used to hold a corresponding control mask. The control can be considered as a field in an entry of the miss queue, however it is accessed using a different associative address from the other fields in the miss queue. Hence, it is held in a separate associative store. The connection from the cache bank to the processors is assumed to have a broadcast capability. The control mask is used to control the broadcasting of data back to the processors.

Each memory request is assigned a tag number. This tag number is necessary to identify each pending request uniquely in all of the miss queues in the system. If an incoming miss reference matches a pending reference's address i.e. they refer to words in the same cache block, it is assigned the pending reference's tag number. Hence, a set of requests to the same block will all have the same tag as the oldest pending request to the block in the queue. The effect of this tagging scheme is that all pending requests to the same block are combined into one memory request. The incoming miss reference's control information, in particular the processor identifier and word index is combined with previously matched reads to form a control mask. Hence, each memory read request to be issued has a corresponding control mask, which records which processors are waiting for the block to be returned from memory, and which word of the memory each of these processors requested. Miss references which have combined with an issued refer-

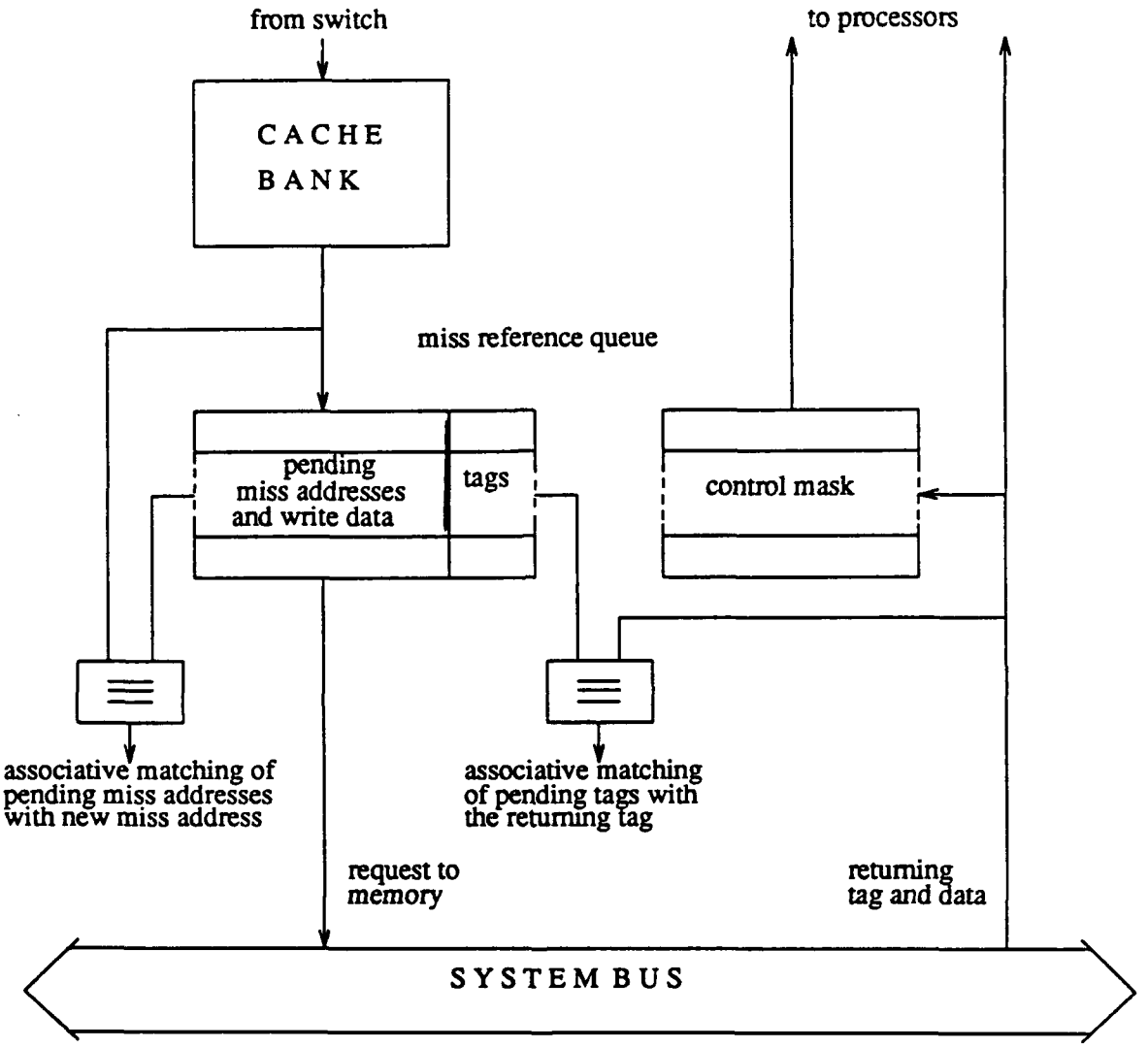


Figure 2.10: Miss queue organization

ence do not have an associated control mask since they do not issue a request to memory.

A new tag is generated and assigned if there is no match. When a pending block returns, it carries the tag associated with the original request. This tag is associatively compared with all

queue entry tags. Miss queue entries with a matching tag are cleared as their requests have been satisfied. The control mask associated with a memory request, is accessed with the tag number of the memory request. The control mask is then used to route the memory data to the appropriate processors and to write the data into the cache.

2.4.3. Write policy

Writes to a cache require two actions. First, the address is checked against the tag store to determine hit or miss; and second, if the reference is a hit, the cache is updated. In our system, we assume that a write reference uses a delayed write update algorithm [FuKH87] so that a cache data access conflict with an immediately following read is avoided. If a read does not reference the same location as the pending write the read can proceed ahead of the write update. However, if the read references the same location as a pending write, the pending write data can satisfy the read request. Only one pending write per cache bank is buffered since the write can always update the cache when the next write arrives in the cache. Thus, cache updates do not disrupt the pipelining of references.

A write reference that misses may or may not cause an update to main memory depending on the memory update algorithm used. There are a number of algorithms for updating cache and memory [HwBr84]. In this study we use a modified write-back algorithm. A write-back algorithm updates the cache but not the memory on a write hit. Hence, copies of a block in the cache and the memory are allowed to be inconsistent after a write. When a write miss occurs, four actions have to be taken. The memory block currently occupying the cache location to be re-allocated is checked to see if it has been updated since it was initially loaded into the cache from memory. If it is dirty, the block must be removed from the cache for write-back before the cache location can be re-allocated to another memory block. In our model, the write-back occurs

immediately, but in other write-back implementations it may be buffered to be completed later. A memory request is made to load the cache with the memory block requested by the original write cache miss. The new memory block is then loaded into the cache and the block is updated with the write data. Thus, in a standard write-back algorithm the write miss address must be held in the cache segment until the block has been loaded from memory. This can potentially result in cache busy conflicts. The advantage of the write-back scheme is that it reduces the memory bandwidth requirements as no writes between replacements need access the memory. With multiple memory requesters that share a cache, as in our system, a write-back scheme is an appropriate algorithm.

We make one modification to the above standard write-back scheme for handling write misses. After the dirty block has been read from the cache for the write-back, the cache moves the write miss address into the miss queue which eventually issues a modified write request to the memory. This modified write causes the memory to read and update the requested block before returning the modified block back to the cache. Thus, the modified write is similar to a read miss except when the block is returned from the memory it is stored in the cache, but no data is returned to the processors. By allowing the memory to do the block update, the original cache miss address and its associated data does not need to be held in the cache segment and they can be moved into the miss queue. There is an extra cost in sending the write data to the memory and having the memory perform read-modify writes. However, this modification simplifies the handling of read-write hazards and reference combining and does not stall the cache pipeline, as discussed below.

The write-back operation itself requires two steps. First, the dirty block is read from the cache into the memory interface and second the bus is requested to send the block back to the

memory. For simplicity, we assume that the memory interface has sufficient buffering to hold the address and data and that the write-back operation has a higher priority than any requests waiting in the miss queue. Furthermore, during the cycles that the cache block is being loaded into the memory interface the cache is unavailable to incoming references.

2.4.4. Read-write hazards and request combining

When pipelining is employed to improve throughput in a way that allows requests to be served out of order, care must be taken to resolve read-write hazards [Kogg81, RaLi77]. There are three types of hazards: read after write (RAW), write after read (WAR) and write after write (WAW). The goal is to resolve these hazards at a minimum cost to performance.

WAW hazards are easily handled given the first-in first-out queuing nature of the miss queues. Consecutive writes to the same block can be combined by merging their write data to give the most up to date write data. Unlike combining reads with reads or reads with an original write miss, a new write cache miss cannot be combined with a pending read or a write miss to the same block.

In a RAW hazard, the read reference following the write must get the updated data. In the standard write-back algorithm, which does not update the memory on a write miss, the read after the write cannot follow the write into the miss queue. The read must stall in the cache segment, so that it can read the cache after the cache update has occurred. However, with the modified write-back algorithm used here, the write miss returns an already modified block to the cache. Thus, the read can be placed into the miss queue where it will be matched with the pending write miss. When the modified block is returned, this read miss is satisfied by the updated information in the returned block.[1] By allowing a write miss to return updated data to the cache, the proposed

modified write-back algorithm is better suited for overlapped miss handling.

The correct resolution of a WAR hazard is that the read must get the data before the update by the write. Therefore, in this instance the two references should not be combined even when they refer to the same block since the write following the read generates a new value for the some memory location in the block, possibly the same location. This restriction on combining results in two memory requests being sent for the same block. The memory data fetched by the read miss is overwritten in the cache by the memory data for the write miss. An alternative would be to issue only one request to the memory. This would require that the write miss either feeds-back to the cache after the data has been returned or is held in the cache store segment until the data has been returned, so that the cache can be updated by the write i.e. turn a write miss into a write hit. Holding the write miss in the cache store segment would increase the chances of a cache bank busy conflict, thereby reducing the arguments for a miss queue. Feeding back the write miss after the read miss has updated the cache is a possible alternative but would require more complex control. Hence, for simplicity we issue two memory requests in our simulation model.

Table 2.1 summarizes which miss references can be combined in the miss queue. All references are assumed to be for the same block. The first column, *miss type*, is the type of cache miss entering the miss queue. The next two columns show the type and status of requests in the queue. The last column indicates whether the two references can be combined. An *issued* request has already been sent to memory and a *unissued* request is waiting to be sent to memory or has been combined. A new read can be combined with any read request, *issued* or *unissued*. Since a write miss returns modified data in our modified write-back algorithm, a new read can

[1] It may be possible in some cases to satisfy a read miss earlier i.e. directly from write miss data held in the miss queue, but this is too complex to be considered here.

also match with *issued* and *unissued* writes. Note that with a traditional write-back scheme the read miss would be stalled in the cache queue or earlier until the write miss memory block was loaded and modified in the cache. A new write miss cannot match with reads or *issued* write requests. A write miss can only match with an *unissued* write miss if the writes are consecutive references. For example, consider an ordered set of 5 miss references, {read, write, write, read, write}, where the leftmost read is the first issued reference, the following write is the next issued reference etc. and all references are to the same block. Assume that the first two references are in the miss queue where the read is *issued* to the memory and the write is *unissued*. These two references do not get matched in our scheme. A write cache miss enters the queue and this gets combined with the *unissued* write miss. The next reference to enter the queue is a read miss and this can be combined with the two combined write misses. Finally a write miss enters, but this does not get combined so that the integrity of the previous combined read is preserved. Hence for this set of 5 references, only 3 requests get sent to memory.

miss type	most recently queued miss to the same block		combine?
	type	issued?	
READ	READ	Y or N	Y
	WRITE	Y or N	Y
WRITE	READ	Y or N	N
	WRITE	N	Y
	WRITE	Y	N

Table 2.1: Combining miss requests

In the above description we have assumed that references are only matched with requests issued by the same cache bank. This matching task is further complicated when memory addresses within the same block are interleaved amongst the cache banks. Interleaving is discussed in the following sub-sections.

2.5. Multiple cache banks

When multiple processors attempt to access a common memory resource there is a potential for conflicts. If the memory system is organized as a monolithic structure, conflicts among accesses may have a severe effect on a processor's performance. For example, consider p processors connected to a shared cache through a $px1$ cross-bar switch. Assume all references are hits and that it takes one cycle each to traverse the switch and cache. If all processors make a reference in the same cycle, then the average time a reference waits for the cache is:

$$\frac{1}{p}\sum_{i=0}^{p-1} i$$

Such a high degree of conflict removes any advantage of using a shared cache organization.

A well known technique used to improve the performance of a common memory system is to organize it as a set of independently accessible modules. The addresses are then distributed amongst the modules in some ordered manner. This distribution of addresses, i.e. address interleaving, can also be applied to a shared cache design if it is organized as multiple banks. While it is one logical cache, it physically consists of multiple banks, each able to support an independent reference stream. Each bank is essentially a separate cache in itself, with its own data and tag store and memory interface. A word reference is routed to the correct cache bank by the cross-bar switch. The potential for concurrent access to several cache banks reduces the degree of

conflicts in the switch and hence the probability of processor pipeline stalls.

2.5.1. Address Interleaving

There are two basic methods for distributing or interleaving the addresses among the memory modules: word level and module level. Consider a memory with a size of $N (=2^n)$ words distributed among M memory module and an address of n bits, i.e. $a_{n-1}...a_0$.

In word level interleaving, a word with an address k is located in memory module k modulo M. The lower $m (=\log_2 M)$ bits are used to select the memory module while the upper $n-m$ bits are used to select the word within the memory module. For $P \leq M$, there are no memory access conflicts if each processor addresses a distinct memory module. Processors that simultaneously address the same module are in conflict. At most one request of a conflicting set can be accepted. For $P > M$, at least one processor must be in conflict with some other processor when at least $M+1$ processors issue a request within a memory cycle.

In module level interleaving, the upper m address bits are used to select the module. Each module consists of $2^{(n-m)}$ consecutive words. As above, no conflicts occur when there are $P \leq M$ requests in a particular cycle and each request addresses a distinct module.

The interleaving technique can also be applied to multiple cache banks. For cache interleaving we consider three schemes: word, block and split level interleaving. There are a number of other schemes to map addresses to memory modules, for example address skewing or randomizing. However, these generally require more complexity which can lengthen the cache access cycle. These schemes are not considered here. The word, block and split level interleaving schemes are depicted in Figure 2.11, for $N=16$, $M=4$, address bits are $a_3a_2a_1a_0$ and there are 4 words per block. The first scheme, Figure 2.11a, is the word level interleaving scheme described above, where the m low order address bits, a_1a_0, are used to select the cache bank. In

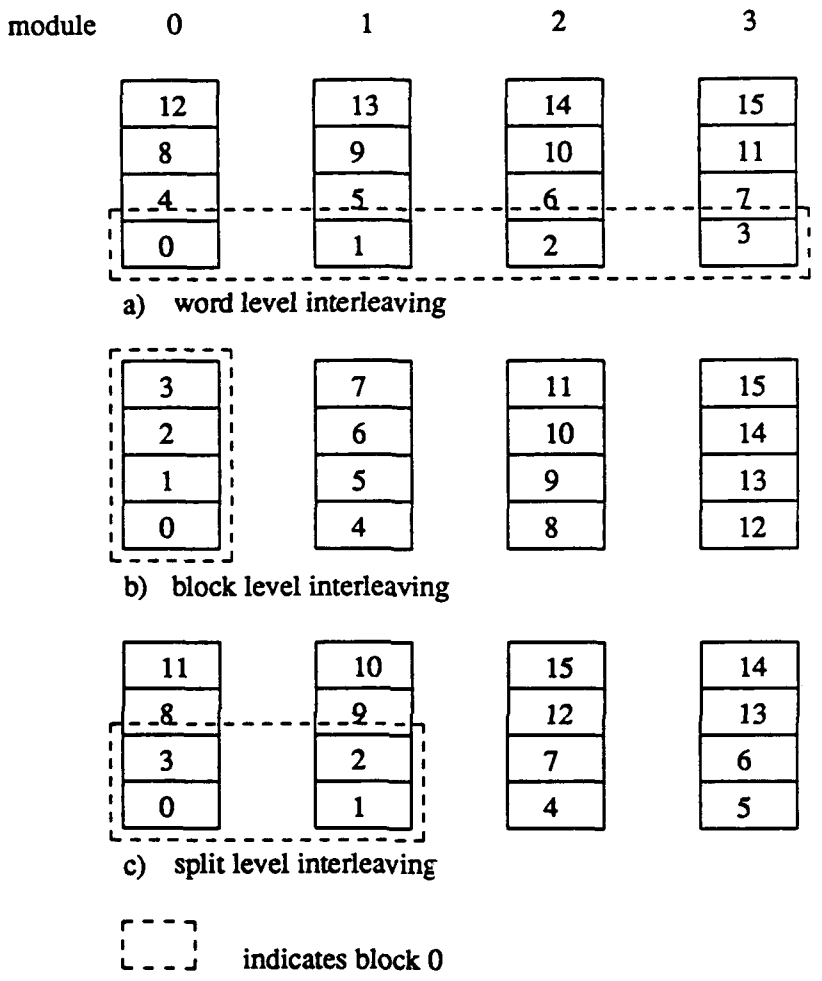


Figure 2.11: Cache and memory interleaving schemes

this scheme each word of the block resides in a separate cache bank for a block size less than or equal to M. The second scheme, Figure 2.11b, takes into account that the unit of mapping in the cache is the block. Each block is located entirely within one cache bank with successive blocks being located in successive cache banks. The address bits, a_3a_2, are used to select the cache bank. These are generally found by skipping over the low order bits used to select the word

within the block and using the next lowest order m bits to select the bank. The third scheme, Figure 2.11c, is a compromise between the two previous schemes. Here a block is word level interleaved over K banks, where K is a power of 2 and $K<M$. For example if $K=2$, the words of a block are always split over two banks. Block interleaving is $K=1$. Word interleaving is $K=M$ if block size $\geq M$. In Figure 2.11c, $K=2$, but an irregular interleaving is used where the odd and even words of the blocks are equally represented in each bank. The cache bank is selected by $a_2(a_1 \oplus a_0)$, while the word within the cache bank is selected by $a_3 a_1$. This form of interleaving is used by the 4 module shared cache of the Alliant FX/8.

2.5.2. The effect of interleaving on the cache banks

Interleaving addresses among the cache banks has three effects of concern here. It dictates first, which address patterns will cause conflicts in the switch, second, which banks are written when data is returned from memory and third, the complexity of the miss queue combining.

The distribution of the words among the cache banks has an effect on cache access latency. For particular access patterns, conflict free access is possible. For example, consider the loop in Figure 2.12a which sums the elements of arrays B and C into array A. Assume that there are four cache banks with a block size $b=4$ words and that each cache bank is one word wide. Let each array be block aligned, starting in cache bank 0 and each single element of an array be of a single word size. Furthermore, assume that 4 processors are available for executing the loop in parallel and that the number of loop iterations, N, is a multiple of the block size b.

If the cache is word interleaved then conflict free access is possible if the iterations of the loop are horizontally spread across the four processors, the cache accesses are all hits and the processors remain in lock step. The index of the array elements accessed by the four processors are as shown in the table of Figure 2.12a. For example, in iteration 0 processor 0 reads B[0] and

```
for(i=0; i < N; i++)
    A[i] = B[i] + C[i];
```

		processors			
		0	1	2	3
iterations	0	0	1	2	3
	1	4	5	6	7
	2	8	9	10	11
	3	12	13	14	15
	⋮	⋮	⋮	⋮	⋮
	N/b	N-4	N-3	N-2	N-1

a) horizontal spreading

```
for(k=0; K < N/b; k++)
    for(i=k*b; i < (k+1)*b; i++)
        A[i] = B[i] + C[i];
```

		processors			
		0	1	2	3
iterations	0	0	4	8	12
	1	1	5	9	13
	2	2	6	10	14
	3	3	7	11	15
	⋮	⋮	⋮	⋮	⋮
	N/b	N-3b	N-2b	N-b	N-1

b) blocking

Figure 2.12: Conflict free access to a shared cache

C[0] and writes to A[0] while processor 1 reads B[1] and C[1] and writes to A[1].

If the cache banks are block interleaved, then horizontal spreading of the loop iterations will result in conflicts in cache access that may degrade system performance. For example in iteration 0, the four processors will all attempt to access cache bank 0. A possible schedule that will result in conflict free access requires each processor to access different blocks in different banks in each cache cycle. One possible approach is to re-write the loop as shown in Figure 2.12b, where the loop is broken into two nested loops and the outer loop is scheduled on different processors while the inner loop causes each processor to access only the words of a particular block. For example in the first 4 iterations for a block size *b* of 4 words, processor 0 accesses words 0, 1, 2 and 3 of each array, processor 1 accesses 4, 5, 6, 7, etc. The index of the array elements accessed by each processor is shown in Figure 2.12b, where the iteration number refers to the inner loop iterations. Processors never conflict with each other regardless of whether there are cache misses or processors fail to maintain lock-step execution.

When a miss reference returns from the memory, how the cache is interleaved determines which banks must be written with the data. Assuming only demand cache references, any returned data should have the highest priority of access to the cache.[2] References attempting to address the cache bank, must be stalled while the returned data is written into the cache bank. In word interleaving, where all words of a block are in distinct banks,[3] each bank is busy for the time taken to write the appropriate word. In block interleaving only one bank will be busy, but it will be busy for the time taken to write all the words of the block.

In section 2.3.2, matching miss addresses with pending memory requests was considered to be important for achieving coherence and a high performance cache design. The interleaving

[2] Data returned for a prefetch miss reference should have a lower priority access to the cache than demand references.

[3] This is true if the number of words in the block is less than or equal to the number of cache banks, and each bank is one word wide.

scheme used in the cache banks affects the complexity of this task. When block interleaving is used, all words in a block reside in the same cache bank and miss addresses need only be matched against the outstanding requests in the miss queue of the same cache bank. Matching is relatively simple as is the information needed to track the combining status.

When a block is split across the cache banks, as in word or split level interleaving, a miss request for a particular block may have been initiated from any of the banks that can hold any words of the block. An incoming miss reference must also be matched against all of the miss queues in all of the cache banks that could have originated a request for that block. To manage the matching of the references correctly, the status of the queues can be maintained in one or more global status buffers which can be accessed by incoming miss references as well as by returned tags. A global status buffer is needed for each set of cache banks that a cache block may be split across and its size is a function of the number of cache banks, the interleaving scheme, and the block size. For b words in a block and c cache banks (where b and c are powers of 2), word interleaving requires $\frac{c}{b}$ global status buffers when $b<c$. Although only one is required when $b \geq c$, it is the same size as the sum of the $\frac{c}{b}$ buffer sizes for the $b<c$ case. For split interleaving (assuming that the block is always split over 2 cache banks), $\frac{c}{2}$ global status buffers are required for $c \geq 2$. If a global status buffer can only be accessed by one request at a time, then miss references issued simultaneously to different banks in the same set must be prioritized for the match look-up. An alternative organization is to have each bank maintain its own separate global status for each block that completely or partially resides in that bank. In this way multiple references could proceed in parallel. However, this scheme requires some complex mechanism for the cache banks in a set to communicate with one another so as to maintain status

coherence. In practice, the complexity of handling inter-bank matching may not allow deep buffering to be employed, but small buffers may be sufficient, assuming that the hit ratios are sufficiently high and miss bursts are sufficiently seldom. For our simulation purposes we have chosen to use the logically simpler global status approach in order to get an initial estimate of performance.

Since all references to the same block, regardless of which bank originated the references, must be matched with the appropriate global status buffer, interleaving does not affect the hazard resolution scheme.

2.6. Memory Interleaving

Interleaving of addresses in the memory system follows the schemes selected for the cache banks. We consider a memory module to consist of a data store and all the hardware necessary for address decoding, data access, bus arbitration, and driving the bus for data return. Assume that each module is one word wide and can only service one request at a time.[4] During the service time, the requested memory module is busy. How the addresses are distributed among the memory modules determines the latency for accessing a block, and the time that particular memory modules remain busy for each cache miss request.

Consider m memory modules in the system and b words in a block, where $b \leq m$. Additionally assume that it takes just one cycle to broadcast a memory request to all the memory modules, that a memory module takes l cycles to access a word, and that a word can be transmitted over a word-wide bus in w cycles.

[4] The number of requests accepted by a memory module can be increased by locating buffers within the module, however this is not considered in this study.

For word level interleaving of the memory modules, the memory request for a block is received by all the memory modules in the same cycle. Each memory module that holds one or more words of the requested block proceeds to access its data. After l cycles, the words are sequentially transmitted to the cache banks through the bus at the rate of w cycles per word. The bus is busy for 1 cycle during the transmission of the request, and it is idle for l cycles and then busy again for bw cycles, during the sending of b words of data. Since $b \leq m$, all words of the block are accessed in parallel from different modules, and b modules are busy for l cycles. For two simultaneous requests to the same set of memory modules, one request must wait for at least $1+l+bw$ cycles to get to the memory.

In block level interleaving all words of a block are located in the same memory module. All b words of the block are accessed sequentially before any of them are transmitted onto the bus. The bus is busy for 1 cycle, then one memory module is busy for bl cycles, and finally the bus is busy again for bw cycles. An alternative scheme that would have a lower latency for block access would overlap the access of some of the words with the transmitting of accessed words of the block. However, this scheme would require each memory module to be pipelined, increasing its complexity. Furthermore, we are interested in allowing other requesters access to the bus. A memory request results in only one memory module being busy, but it is busy for bl cycles. However, during this memory module busy time, the bus and other modules are accessible by other requesters. For two requests to the same set of modules, one request must wait for at least $1+b\,(l+w)$ cycles to get to the memory.

The tradeoff between the two schemes is in memory access latency during concurrent access. With word level interleaving more memory modules are busy, but the access latency of an isolated request is shorter. With block interleaving the latency is longer, but only one module

is busy on an access, thereby allowing greater concurrency in serving miss requests. The split level scheme is a compromise between word and block interleaving.

Figure 2.13 shows reservation tables for the word, block and split interleaving schemes for 4 memory requests with m=4, b=4, l=2 and w=1. The tables show the cycle status of the four memory modules and the shared bus. Note that in all the schemes the return of accessed data from the memory has the highest priority for use of the bus.

2.7. Bus protocol

Bus protocols can be separated into two types: locked and interleaved. This classification refers to the state of the bus for transactions. A transaction is defined as a request and the satisfaction of that request. For instance, a read transaction starts with a read request and ends with receiving the data. In a locked bus, once the processor has gained the bus and sent the request, the bus is locked to other requests until the completion of the transaction. A locked bus has the lowest complexity and hardware costs, but due to locking, it tends to give the lowest performance. Such a system is unsuitable when there are r tiple requesters and multiple memory modules.

An interleaved bus protocol supports multiple active transactions. A request uses the bus only to complete a specific activity. Thus there can be several simultaneously pending memory requests. For instance, in a read transaction the bus is used once to transfer the read command and again later to transfer the data. With an interleaved bus, the time between the two activities is available for activities of other transactions, as in Figure 2.13(b) and (c). An interleaved bus has no benefit in Figure 2.13(a) since all memory modules are busy while the bus is free. Since sending the request and its response are separated and may be interleaved with other requests and responses on a interleaved bus, it is necessary to provide a method to complete each

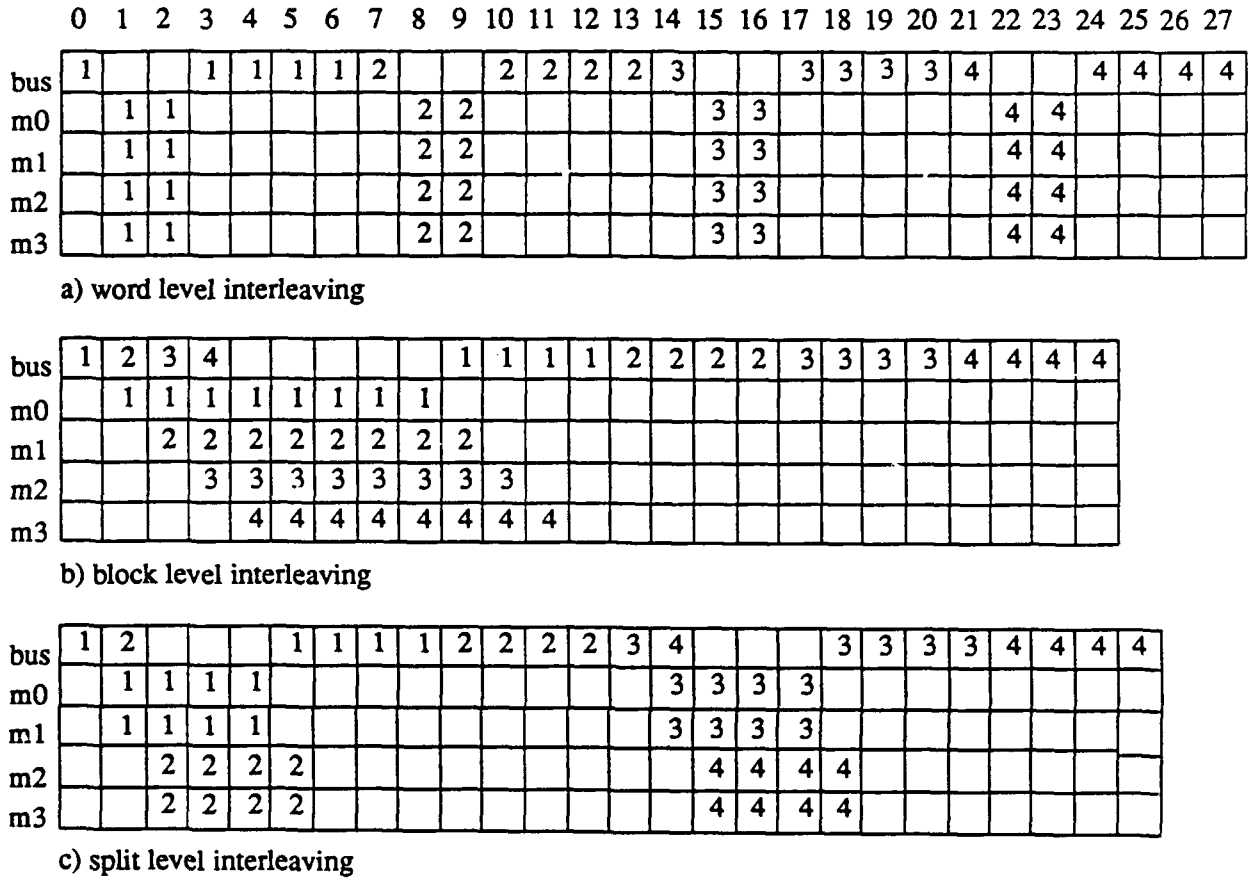


Figure 2.13: Bus and memory access for word, block and split interleaving schemes

(m=4, b=4, l=2, w=1)

transaction correctly. A simple scheme is to tag all requests and responses with an identifier which is then used to route the data correctly to the original requesters. An interleaved bus is used in the system under study. We use the cache bank number and the queue position of the original request as the identifier to route the return data. This identifier is also known to all requests that are combined with the original miss request.

CHAPTER 3

SIMULATION ENVIRONMENT

To evaluate the effectiveness of the design choices discussed in Chapter 2, a simulation model has been developed. In this Chapter we briefly describe the model and its operation.

3.1. Simulation model

The simulation model described in this Chapter has been implemented using the CSIM environment [Schw86] and is shown in Figure 3.1. The model simulates a set of p pipelined processors connected to a shared cache of c banks through a pxc cross-bar switch. The cache banks are all connected through a shared bus to the memory modules. Each processor is modeled as a two segment pipeline. A pipeline segment takes one pipeline cycle to complete its operation and all timings are in units of pipeline cycles. Each processor can execute an instruction stream independently of the others and has support for vector instructions.[1] A vector is defined as an ordered set of related memory locations whose consecutive addresses are separated by a constant known as the stride. A processor makes vector references as a set of consecutive scalar references issued, unless there are conflicts, in consecutive pipeline cycles. We assume that each processor can take full advantage of pipelining and can overlap these references subject only to available resources. A processor queue is located between each processor and the switch, a cache queue is located between the switch and each cache bank and a miss queue is located between each cache bank and the shared bus. Note that the memory interface is implicit in the miss

[1]The simulation results presented in Chapter 4 are limited to the performance of the system for parallel execution of vectorized code only.

queues. That is, the memory interface functions, such as constructing the memory request and arbitrating for the bus, occur at the same time that the miss queue is being loaded. Furthermore the global status buffers, used to perform miss queue matching in word and split level interleaving, are also assumed to operate while the miss queue is being loaded.

We assume that each reference issued to the cache is for a 64-bit word. A cache bank is 8 bytes (one double-word) wide and the block size is an integer number of double words i.e. a

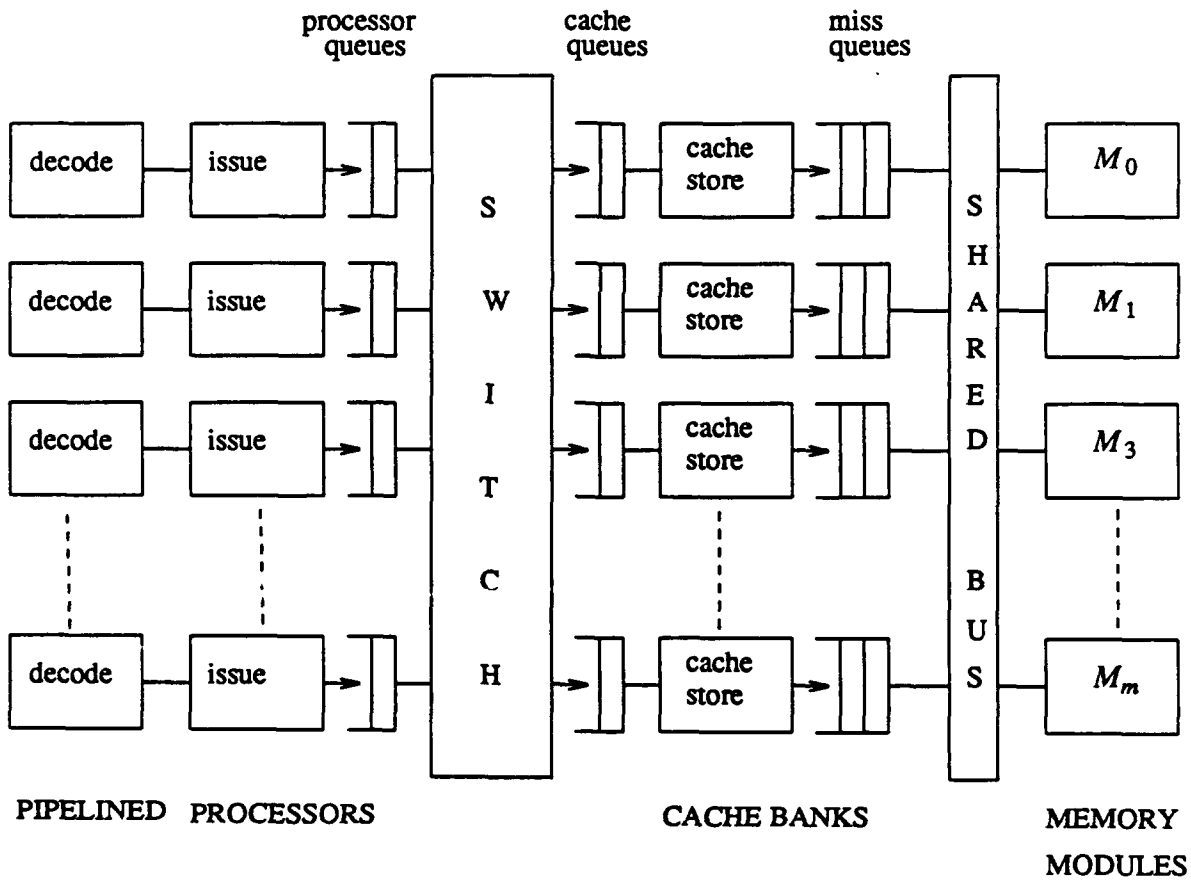


Figure 3.1: Multi-processor with shared cache and buffering

block size of 4 is 32-bytes. Each memory module is also one double word wide.

There are fourteen parameters that can be varied in the simulation model. These parameters are: number of processors, three queue depths (processor, cache and miss queue), cache size, number of cache banks, cache block size, number of memory modules, number of cycles for a pipeline segment, for memory access, for sending a memory request on the bus, and for returning a block of memory data on the bus, and cache and memory interleave schemes. To simplify the choice of simulation parameters we define a base system as shown in Table 3.1. The value *segment time* is the minimum number of time units taken to pass through a pipeline segment, the switch, a cache segment, a processor or a miss queue if there are no stalls or conflicts. The values *send request*, *memory cycle* and *return data* are the number of time units taken to send a memory request on the bus, access a double word from a memory module, and return a block from memory on the bus, respectively.

In the simulation experiments one or more simulation parameters are changed at a time, with respect to the base system, to see their effect on performance.

3.1.1. Address traces

The simulation model is driven by address trace files. The traces were generated by a detailed simulation of program execution on an Alliant FX/8 multi-processor. The output from the simulator is a sequence of memory references. For each reference the simulator generates the memory byte address, the number of bytes to be accessed, the estimated number of cycles since the last reference, and the type of reference i.e. scalar read, vector write etc. The estimated number of cycles between references is used to determine the time that references enter the simulation model. To produce the traces, routines were compiled with the Alliant FX/Fortran parallelizing and vectorizing compiler and executed on the simulator. Primarily we are

interested in the system's behavior for vector references in fully parallel loops, since these make the most severe demands on the memory. Therefore the routines selected have a very high degree of concurrency and vectorization. The traces were processed and compressed by a filtering program to extract only vector memory references. This was done because:

(1) Instruction fetches will have a minimal effect on the cache for two reasons. Firstly the loops we are simulating are necessarily small compared to the data size. Second, typical systems have a separate instruction cache with a hit ratio that approximates 100%. Assuming no instruction references to the cache in the model is thus reasonable.

(2) Scalar accesses for the selected routines were generally clustered together and scheduled on one processor with the other processors being idle. These references are thus not subject to the contentions associated with the shared cache organization and their inclusion in the simulations would bias the results.

The routines used in generating the results presented in Chapter 4 are listed in Table 3.2. For each routine the dominant stride and length of vector accesses are noted except for UXF where no stride or vector sizes dominated. All programs were compiled to execute on eight processors with full parallelization and vectorization enabled. The routines LFK8, LFK14, LFK15 and LFK18 are routines taken from the Lawrence Livermore set of Fortran Kernels [Mcma86].

procs	ref/cache/miss queue depths	cache size/banks	block size	memory modules	segment time	memory cycle	send request	return data	interleave cache/memory
8	1/1/2	64Kbytes/4	4	4	1	4	1	2	word/word

Table 3.1: Base system parameters

Routine	#references	source and function	stride & vectors
LFK8	226306	Livermore Fortran Kernel loop 8	stride: 5 & 1; vector: 32
LFK14	122560	Livermore Fortran Kernel loop 14	stride: 8 & 1; vector: 8,4
LFK15	78048	Livermore Fortran Kernel loop 15	stride: 1; vector = 32
LFK18	212736	Livermore Fortran Kernel loop 18	stride: 4; vector = 16
BTRIX	259403	ARC3D, computationel fluid dynamics	stride: 1,5,8; vectors: 8,16,32
NEWRZ	153335	SIMPL2, computation in hydrodynamics	stride: 1; vector: 30,31,32
RRK	217119	kernel of experimental physics	stride: 1; vector = 32
UXF	59650	VORTEX, PDE solver	various strides and short vectors

Table 3.2: Benchmark routines

The other routines were taken from various application programs gathered at the Center for Supercomputing Research and Development. No attempts were made to optimize the routines for the organizations simulated.

3.1.2. Operation

The model simulates eight parallel processors issuing references to a shared cache. A reference is read from the trace file and passed to a processor at the time associated with the reference, if the processor is available i.e. the processor is not stalled in its D segment. Since the trace file only contains memory references, the time, t between references can be modeled by inserting $t-1$ as non-memory instructions as discussed in Section 2.2

Each processor can place a reference into its processor queue whenever the queue is not full. If a processor attempts to issue a reference when its processor queue is full, it must stall until a queue entry becomes available. The processor queues feed into the switch which routes request to the requested cache bank. Selection through the switch is on a first-come-first-served basis with the processor number being used for arbitration when necessary. If more than one

processor has a head of queue reference destined for the same cache bank in the same switch cycle and that cache bank is available, the reference issued by the highest priority processor is selected to be routed through to the cache bank while the other references must remain in their processor queues until the cache bank becomes available.

A reference takes 1 pipeline cycle to pass through the switch and to address the cache in the next pipeline cycle. Note, if a cache queue exists, winning the bank does not necessarily mean the cache data store is accessible immediately. It only means that at least one queue entry is free. However, entering the cache queue does free the switch for further references.

If a reference hits in the requested cache bank, the cache bank becomes available in the next cycle to any reference waiting at the head of a processor or cache queue. If a reference is a cache miss and a miss queue entry is free, the reference is moved into the miss queue where it may be matched with pending memory requests. If word or split interleaving is selected for the cache banks, then a global status buffer scheme is used for memory request matching. A block interleave scheme only requires matching within the relevant cache bank. To simplify the model we assume that, regardless of the interleave scheme, matching is not in the critical path and that it can be accomplished, in some fashion, before the memory request attains the shared bus. References that match in the queue do not generate requests to memory as the requests can be satisfied by some already issued memory request. However, such combined requests do use a queue entry.

Moving cache misses into the miss queue makes the cache available to the next reference as in the case of a cache hit. If no entries in the miss queue are available then the reference must remain in the cache bank blocking any subsequent requests to that bank until the miss queue is no longer full.

A request is issued by a cache bank to the memory modules only if it was a cache miss and it did not match a previously pending request in the miss queues. In issuing a memory request, a cache bank must compete with other cache banks for the shared bus. A cache bank that obtains the bus but finds the requested memory bank busy must release the bus for the next highest priority cache bank. A request to a memory module takes one time unit on the bus. A memory request is held in the miss queue until the requested data has returned. Writing requested miss data into the cache has the highest priority for cache access and may result in cache access conflicts that may delay new requests incoming from the switch. For simplicity we have assumed that updating the cache takes place in the same cycle that the data is sent on the bus. Thus updating a cache bank takes two consecutive pipeline cycles.

A write-back request to the memory occurs if a dirty cache block is to be re-allocated. If it takes n cycles to send the block from the memory to the cache, we assume it takes n cycles to "unload" the cache block into the memory interface and then n cycles to send the block back to the memory. During these n cycles the cache is unavailable for access, but the write miss address is moved into the miss queues. The write-back operation is assumed to have the highest priority for the bus within the cache bank. The cache becomes immediately available for access after n cycles. In the base system we use a value of n=2.

Figure 3.2 shows the timing for 6 references. A H reference is one that will hit in the cache and a M reference will miss. The subscript is the processor issuing the reference, the superscript is the number of the reference being issued by that processor and (m,n) denotes that block m in cache bank n is being accessed, e.g. $H_3^1(b,y)$ is the second reference made by processor 3 and block b of cache bank y is being accessed. The first five references ($M_0^0(a,y)$, $H_1^0(b,y)$, $M_2^0(a,y)$, $H_3^0(b,y)$, $H_3^1(b,y)$) are all to the same cache bank, whereas $H_1^1(a,z)$ is to a different

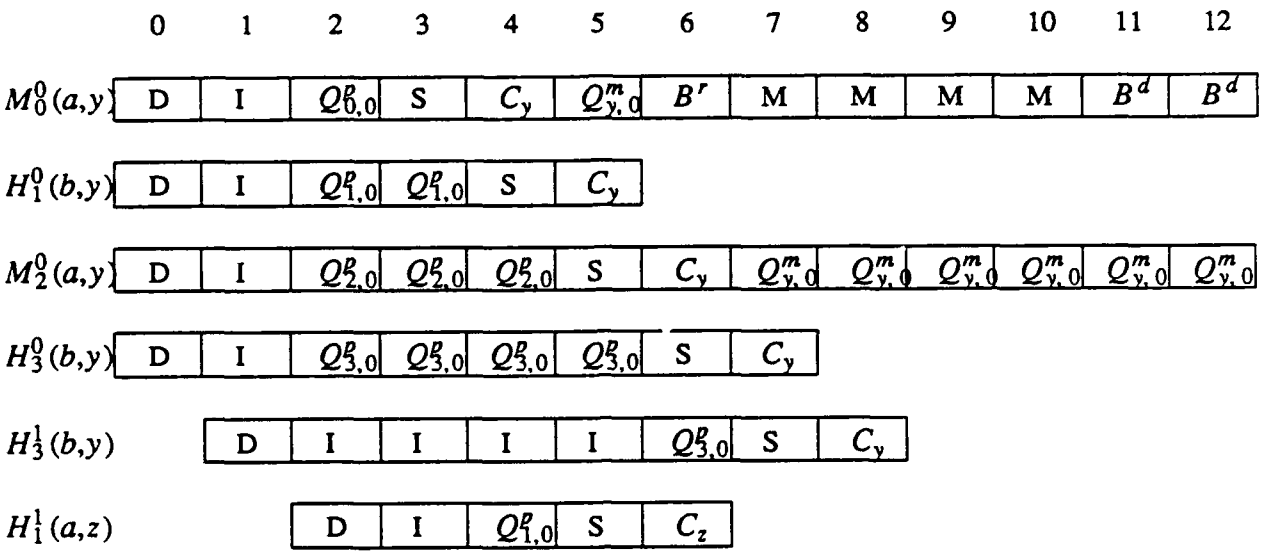


Figure 3.2: Timing diagram for reference processing

bank. Reference $M_0^0(a,y)$ is a cache miss and a memory request is issued to the memory system. The next reference $H_1^0(b,y)$ is a cache hit. As the reference is of a lower priority than $M_0^0(a,y)$, it must wait in the processor queue of processor 1 for an additional cycle before it passes through the switch. However it is not delayed from addressing the cache by the previous miss. The following reference, $M_2^0(a,y)$, is also a cache miss but it is to the same block as the pending request to memory. This reference is passed to the miss queue where it combines with the previous miss reference. It is therefore satisfied at time 12 when the first miss reference is completed, and is removed from the miss queue at this time. The next two references, $H_3^0(b,y)$ and $H_3^1(b,y)$, are

both issued by processor 3 and are both hits. As there is only a single position in each processor queue, the second of these references cannot be moved into the processor queue and stalls the issue stage of the pipeline until the queue is available at time 6. The last reference, $H_1^1(a,z)$, is issued at time 2 and is the second reference from processor 1. The reference is to a different cache bank from the previous references and thus avoids conflicts and completes in the minimum number of cycles. Note that, had $H_1^1(a,z)$ been issued at time 1, then it would have stalled in the issue stage for one cycle as processor 1's previous reference, $H_1^0(b,y)$, is held in the processor queue until time 4.

3.2. Simulation performance indices

In this section we discuss the performance indices used to present the simulation results in the next Chapter.

3.2.1. Completion time

One measure we use is the overall reference completion time. As we are essentially interested in the performance of the system in accessing data from the cache and the memory, the completion time in this study refers to the time at which a particular processor receives its last requested data item from the cache or the memory. Furthermore, since we are simulating parallel programs on a parallel processor, the system completion time is defined as the time at which all processors have completed all their references.

3.2.2. Cache hit ratio as a performance index

The cache hit ratio has been used as the primary performance index in a number of previous architecture studies involving cache memories. The hit ratio is the number of references that hit in the cache divided by the total number of references issued to the cache. The hit ratio is

strictly a measure of the fraction of all data references that is found in the cache without accessing the memory. Using only the hit ratio as the index for the cache-related performance of a system may be quite misleading. Mechanisms may exist, as part of the cache or processor design, that can reduce the performance degradation due to cache misses. For example, a low hit ratio in cache may be acceptable if a very fast load path from memory is available, or if a prefetch mechanism can access the cache early, before the data is actually needed, so that the latency of a miss can be partially or fully hidden. Such mechanisms are not accounted for in the cache hit ratio.

In our model the cache hit ratio is an inadequate performance index for the following reasons:

(1) Performance degradation is due not just to the cache hit ratio but also to the cross-bar switch and cache conflicts.

(2) The role of the processor and cache queues is to decouple, at least partially, the performance of the switch and the cache from the processor.

(3) Miss references are pipelined to the memory to increase throughput, miss buffering allows multiple misses to be handled concurrently in the cache, and new requests can be served while misses are queued.

(4) The latency for accessing the cache is indeterminate due to cache and cross-bar conflicts.

(5) The latency for accessing a block from memory is indeterminate due to contention on the shared data bus and at the memory module.

(6) With queue matching and combining, a cache miss may not even generate a memory request. A cache access results in one of three possible actions. A cache hit does not use the miss pipeline and has the shortest average access latency. A cache miss that generates a

memory request has the longest average latency. A cache miss reference that combines with a pending memory request can have an average access latency nearly as short as a cache hit or nearly as long as the miss request that generated the memory request with which it combined.

The hit ratio is an adequate indicator of performance if the processor is tightly coupled to the performance of the cache and the cache, in turn, is tightly coupled to the performance of the memory, that is, if the processor does not issue overlapped references, the cache does not pipeline miss references, and the latency for a cache hit and a cache miss are predictable. These characteristics are incompatible with computer architectures that use pipelining and parallelism to achieve high performance. In the next section we discuss the use of stall and wait cycles as a performance index.

3.2.3. Stalls and waits as a performance index

Ultimately, the performance measure for the system is the rate at which data requests can be satisfied. A performance index for the system under study must take into account the effect of increasing the queue depths and the number of cache banks. While the completion time provides an overall indication of system performance it does not show, in detail, how increases in the queue depths and number of cache banks affect data accesses. In the shared cache multiprocessor, references are subject to conflicts at the switch and at the shared bus which will result in additional or *penalty* cycles to the minimum data access latencies. Hence, a possible measure of the effectiveness of a particular system configuration is the number of penalty cycles it imposes on references.

In our system, a reference can be prevented from advancing in the processors and the cache banks as the result of stalls and waits, as shown in Figure 3.3. A reference is in a stall cycle if it is blocked from being moved to the next pipeline segment due to the lack of a needed resource. During the stall, the prior segments in that pipeline are also stalled. There are two stall types in the system; processor stall and miss stall. A processor stall occurs when the processor queue is full and the processor attempts to issue another reference into the processor queue. The stall ends when a processor queue position becomes available. A miss stall occurs when there is a cache miss and the miss queue is full. The cache bank is busy during the miss stall. The miss stall ends

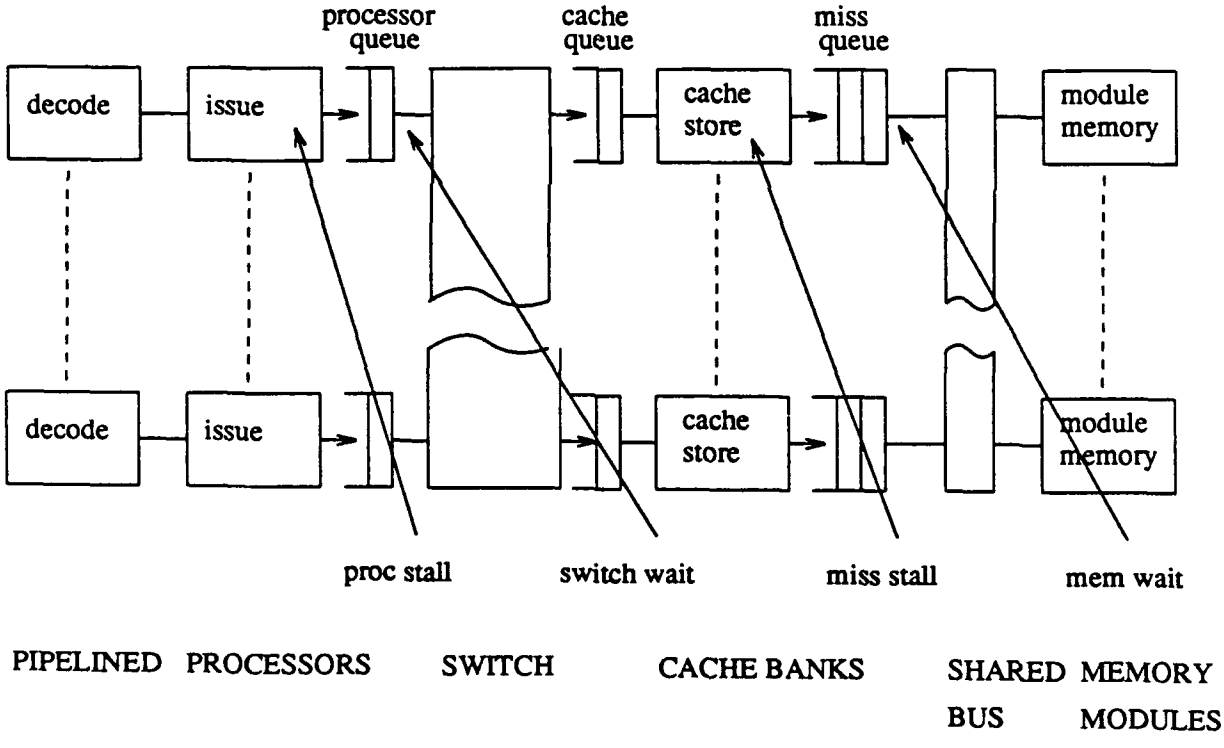


Figure 3.3: Stalls and waits in processors and cache banks

when one or more pending miss references are satisfied and at least one miss queue position becomes available.

A wait cycle is defined as a cycle during which a reference is held in a processor or miss queue. The prior segments are not necessarily stalled during a wait cycle. There are two wait types in the system; switch wait and memory wait. A reference waits for the switch in the processor queue if the cache queue is full as a result of a busy cache bank or there is a switch conflict due to a higher priority reference to the bank being requested. A memory wait occurs when miss requests wait in the miss queue to be satisfied by the memory as defined in detail below.

For performance purposes we focus on the cycles that a reference spends in the system above the minimum number. For instance, *proc stall*, *switch wait*, and *miss stall* are the number of cycles, above 1, that a reference spends in the issue, processor queue and cache segment, respectively. For references going out to memory, the minimum time that a reference spends in the miss queue is the time to send a request to memory, cycle the memory, and return the data. This minimum time occurs if the request wins the bus immediately and the requested memory module is not busy. In this case the reference has 0 *mem wait* cycles. Hence, memory wait cycles are the cycles that a memory request spends in the miss queue waiting for the bus and for the memory module to become available. It does not include the time spent transmitting the request on the bus, accessing the data in the memory, and transmitting the return data on the bus.

If a reference hits in the cache, its average number of penalty cycles is:

$$\overline{t_{pen1}} = \overline{t_{pstall}} + \overline{t_{swait}}$$

where $\overline{t_{pstall}}$ and $\overline{t_{swait}}$ are the average cycles spent stalled in the issue segment for a processor

queue position to become available and waiting in the processor queue for the switch, respectively. If all the hit references have no switch conflicts then $\overline{t_{pen1}}=0$.

References that miss in the cache and get sent to memory incur the following average number of penalty cycles:

$$\overline{t_{pen2}}=\overline{t_{pen1}}+\overline{t_{mstall}}+\overline{t_{mwait}}$$

where $\overline{t_{mstall}}$ and $\overline{t_{mwait}}$ are the average number of cycles that the reference stalls in the cache segment for a position in the miss queue to become available and waits in the miss queue for more than the minimum possible time for memory data, respectively. If the miss queues had infinite depth and the memory system was always able to service a memory requests then the processor model would continue issuing requests at its maximum (program limited) rate, and $\overline{t_{pen2}}=0$.

Due to the matching mechanism not all miss references get sent out to the memory. Matched references have the following average number of penalty cycles:

$$\overline{t_{pen3}}=\overline{t_{pen1}}+\overline{t_{mstall}}+\overline{t_{\delta}}$$

where $\overline{t_{\delta}}$ is the average number of cycles that a matched request waits in the miss queue to be satisfied by an earlier memory request with which it has combined. Let R_1 and R_2 be two miss references that match and combined in the miss queue, with R_1 being the first of the two miss requests to enter the miss queue. If R_1 does not match with any previous miss requests then their respective waits are t^1_{mwait} and t^2_{δ}. If R_2 enters the miss queue in the same cycle that R_1 receives the last returning memory data then $t^2_{\delta}=0$. If there were no bus or memory module conflicts then $t^1_{mwait}=0$ for R_1. If R_2 enters the miss queue in the next cycle after R_1 enters the

miss queue then $t_{\delta}^{2}=t_{mwait}^{1}$. In general t_{δ}^{2}= the number of t_{mwait}^{1} cycles during which R_2 resides in the miss queue.

The number of average penalty cycles per reference is:

$$\overline{t_{pen}}=\overline{t_{pen\,1}}+(1-h)*\left[\overline{t_{mstall}}+q*\overline{t_{\delta}}+(1-q)*\overline{t_{mwait}}\right]$$

where h is the cache hit ratio and q the match ratio in the miss queues, and $\overline{t}$ is an average value of the corresponding t term per request of the appropriate type, i.e. $\overline{t_{pen\,1}}$ is averaged over all the requests, $\overline{t_{mstall}}$ over all miss requests, $\overline{t_{\delta}}$ over all matched miss requests and $\overline{t_{mwait}}$ over all unmatched miss requests.

In the next Chapter we present the simulation results using the above performance indices.

CHAPTER 4

SIMULATION RESULTS

In this Chapter we discuss the results of some experiments performed with the simulation model and the address traces.

To evaluate the effect of individual parameters, the initial set of experiments changes one simulation parameter of the system at a time, relative to the base system. The parameters changed are the processor and cache queue depths, number of cache banks, miss queue depth and interleave scheme. From these results we look at the effect of collective changes to the base system. Finally we look at the cache hit ratio and stalls and waits as indices of performance.

4.1. Presentation of the results

The majority of the results are presented in the form shown in Tables 4.1-4.4. These tables are organized by simulation parameter. The first column indicates the parameter being investigated and its values. All other system parameters unless specified are defined by the base system.

The next column shows the system completion time for each simulation normalized to the completion time for the base system. The completion time is the time at which the last reference is returned to its requesting processor. This time is normalized so that the completion time for the benchmark program under investigation is 1.00 for the base system. This normalized time is also referred to as the overall system performance and $\frac{1}{norm.\ comp\ ıim}$ is the speedup.

The hit ratio is the measured hit ratio of the cache as a whole and is measured over all issued cache references.

The next four columns show the stall and wait cycles, as discussed in Chapter 3, for each configuration. As discussed in section 3.2, *proc stall*, measured by $\overline{t_{pstall}}$, and *switch wait*, measured by $\overline{t_{swait}}$, are measured by the average number of cycles that references stall in the processor queue and wait for the switch, respectively. *Miss stall* is the average number of cycles that references stall the cache bank waiting for a free miss queue position and is calculated as $(1-h)*\overline{t_{mstall}}$, where h is the hit ratio and $\overline{t_{mstall}}$ is the average number of stall cycles for a miss request. *Mem wait* is the average number of wait cycles that a memory request waits for the bus and memory to be become available and is calculated as $(1-h)*(q*\overline{t_{\delta}}+(1-q)*\overline{t_{mwait}})$ where q is the miss queue match ratio and $\overline{t_{\delta}}$ and $\overline{t_{mwait}}$ are the average number of cycles that references wait for the memory, for matched and unmatched references, respectively. The last column shows the average number of penalty cycles, $\overline{t_{pen}}$, that a reference incurs in the system. The calculation for $\overline{t_{pen}}$ is the sum of the four stalls and waits as presented in section 3.2.

4.2. Processor and cache queues

As previously stated, the role of the processor queues is to reduce processor pipeline stalls due to conflicts and cache miss effects. To examine their effectiveness, we vary the queue depth over a range of 1 to 32 entries. In these experiments we only consider varying the depth of the queues between the processor and switch. All other system parameters are fixed at the base system values as shown in Table 3.1 i.e. 1 cache queue position per processor, 4 cache banks, 2 miss queue positions per cache bank and 4 memory modules. The results of the simulations are shown in Table 4.1.

For all the routines, the effect on completion time is 11% or less, over the full range of queue depths. Routines with a high hit ratio (greater than 90%) show a consistent monotonic behavior, as oppose to ones with low hit ratios (i.e. RRK and LFK8). In these high hit ratio cases, *cp tim* decreases for each increase of the processor queue depth. The vast majority (90% or more) of the references are being satisfied at the cache and the processor queues are able to decouple the processor pipeline from some of the switch conflicts and cache miss effects. For hit ratios greater than 90%, *proc stall* sharply decreases when the queue depth is increased from 1 to 32. For LFK14 (~99.4% hit ratio) *proc stall* declines to about 10% of its original value. As the

NEWRZ								LFK18						
proc. queue	norm. cp tim	hit ratio	$\overline{t_{pstall}}$	$\overline{t_{swait}}$	miss stall	mem wait	$\overline{t_{pen}}$	norm. cp tim	hit ratio	$\overline{t_{pstall}}$	$\overline{t_{swait}}$	miss stall	mem wait	$\overline{t_{pen}}$
1*	1.00	93.9	0.89	0.78	0.14	0.01	1.85	1.00	99.0	0.42	0.39	0.04	0.00	0.86
2	0.99	93.9	0.85	0.79	0.13	0.02	1.82	0.98	99.0	0.36	0.42	0.04	0.01	0.83
4	0.97	94.1	0.79	0.83	0.13	0.01	1.79	0.95	99.0	0.24	0.45	0.04	0.00	0.74
8	0.94	94.2	0.69	0.92	0.13	0.01	1.78	0.93	99.0	0.14	0.57	0.04	0.00	0.76
16	0.92	94.8	0.61	1.14	0.14	0.02	1.94	0.93	99.1	0.13	0.60	0.04	0.00	0.78
32	0.90	94.5	0.51	1.10	0.12	0.02	1.78	0.92	99.1	0.10	0.58	0.04	0.00	0.73
RRK								LFK15						
1*	1.00	61.6	10.07	9.41	1.67	0.29	21.66	1.00	90.3	0.95	0.79	0.22	0.02	2.01
2	0.99	62.6	10.15	9.53	1.61	0.30	21.80	1.00	90.2	0.98	0.85	0.22	0.02	2.11
4	1.00	62.9	10.32	9.67	1.61	0.25	22.07	0.98	90.4	0.90	0.89	0.22	0.02	2.07
8	0.99	63.0	10.79	9.72	1.60	0.26	22.58	0.95	90.2	0.70	0.87	0.21	0.02	1.84
16	1.00	62.9	10.90	9.83	1.67	0.26	22.88	0.93	90.3	0.58	0.96	0.21	0.02	1.81
32	1.00	62.8	10.89	9.88	1.60	0.26	22.85	0.89	90.1	0.40	1.09	0.19	0.02	1.74
BTRIX								LFK14						
1*	1.00	97.0	1.20	0.88	0.17	0.04	2.32	1.00	99.4	0.57	0.68	0.03	0.00	1.28
2	0.98	97.0	1.16	0.89	0.17	0.04	2.29	0.98	99.4	0.34	0.69	0.03	0.00	1.06
4	0.96	97.0	1.07	0.93	0.18	0.03	2.24	0.96	99.4	0.17	0.77	0.03	0.00	0.97
8	0.94	97.0	0.99	1.01	0.17	0.03	2.23	0.96	99.4	0.10	0.86	0.03	0.00	0.99
16	0.92	96.9	0.96	1.10	0.18	0.04	2.30	0.95	99.4	0.07	1.14	0.03	0.00	1.24
32	0.91	96.9	0.89	1.13	0.18	0.04	2.27	0.95	99.4	0.06	1.37	0.03	0.00	1.46
UXF								LFK8						
1*	1.00	99.1	0.15	0.14	0.04	0.00	0.34	1.00	81.7	2.62	2.41	0.48	0.07	5.67
2	1.00	99.2	0.09	0.15	0.04	0.00	0.29	0.98	81.4	2.54	2.38	0.46	0.07	5.53
4	1.00	99.1	0.06	0.16	0.04	0.00	0.27	1.02	80.3	2.79	2.73	0.52	0.06	6.20
8	1.00	99.1	0.05	0.17	0.05	0.00	0.28	0.96	82.9	2.58	2.66	0.47	0.06	5.85
16	1.00	99.2	0.03	0.18	0.05	0.00	0.27	0.93	81.3	2.37	2.86	0.45	0.05	5.82
32	0.99	99.1	0.00	0.24	0.05	0.00	0.30	0.92	81.0	2.38	3.14	0.48	0.05	6.15

* base system

Table 4.1: Increasing processor queue depths

number of misses increase, the queue depths become less effective. For LFK15 (90.3%-90.1% hit ratio) *proc stall* remains relatively constant until greater than 4 entries are used. For LFK8 and RRK, which have very low hit ratios, the queue depth has only a slight effect on *proc stall*. In these routines, the high number of cache misses are causing enough conflicts in the switch to keep the queues full regardless of the queue depth. The result is that the system becomes much more influenced by the cache pipeline performance and the system behaves less consistently as a function of the processor queue depth.

One effect of queuing references at the switch is an increase in the inter-arrival rate of references at the switch. This increase increases the probability of switch conflicts which explains the increase in *switch wait* as the queue depth increases. Interestingly, $\overline{t_{pen1}}=\overline{t_{pstall}}+\overline{t_{swait}}$ does not continue to decrease as the queue depth increases. For instance, for NEWRZ, $\overline{t_{pen1}}$ decreases until a queue depth of 8 is reached. At a queue depth of 16, $\overline{t_{pen1}}$ increases, but it drops back again at a queue depth of 32. While the increased processor queue depth allows the processor to increase its reference issuing rate, the switch and cache is unable to accommodate this potential increase in bandwidth fully. Note that $\overline{t_{pen1}}$ forms the major component of the total $\overline{t_{pen}}$ penalty cycles, which exhibits the same sort of non-monotonicity as $\overline{t_{pen1}}$. However, the *norm. cp tim* consistently exhibits monotonic behavior, except for RRK and LFK8 as noted. In this experiment $\overline{t_{pstall}}$ is a better indicator of performance than $\overline{t_{pen}}$, i.e. it is better correlated with *norm. cp tim.*

As expected, the queue depth does not generally have a significant effect on the cache pipeline as indicated by the relatively constant *miss stall* and *mem wait* over all or parts of the queue depth range.

4.2.1. Cache queues

In the discussion of system organization, we mentioned that queues could be located between the switch and the cache. To evaluate the effect of these queues, we increase the depth of the cache queues from 1 to 32 relative to the base system. The results are shown in Table 4.2. It should be noted that processor queues are associated with processors, and cache queues with cache banks. Recall that a base system has 8 processors and 4 cache banks. An experiment with 2 entry cache queues, has a maximum total buffering capacity of (4*2=) 8 requests, while experiments with 2 entry processor queues can buffer (8*2=) 16 requests. Hence, a system with depth n processor queues has the same maximum buffering as a system with depth $2n$ cache queue.

The results generally indicate that increasing the cache queue depth has more effect on the overall completion time than increasing the processor queue depth particularly for RRK and LFK8 where the hit ratio is low. As discussed in Chapter 2, queues between the switch and cache reduce some blocking. This reduction is evident by noticing the generally reduced *switch wait*. Although increasing processor queue depth increased *switch wait* significantly, increasing the cache queue depth decreases switch wait and results in increased switch throughput. However, greater cache queue depth, increases the reference inter-arrival rate to the cache bank and generally increases *miss stall*. RRK and LFK8 show non-monotonic behavior in *miss stall*. LFK14 remains consistent and others show some minor non-monotonicities.

The increase in cache queue depth results in dramatic increases in performance for LFK8 and RRK, where the hit ratio is very low. The reason is that the performance of these two routines are most influenced by the cache misses. The cache queues decouple the switch from this effect much better than the processor queues. As the cache queue is located between the switch and the cache, the result, unlike the processor queue depth increase, is a decrease in *switch wait*.

NEWRZ								LFK18						
cache queue	norm. cp tim	hit ratio	$\overline{t_{pstall}}$	$\overline{t_{swait}}$	miss stall	mem wait	$\overline{t_{pen}}$	norm. cp tim	hit ratio	$\overline{t_{pstall}}$	$\overline{t_{swait}}$	miss stall	mem wait	$\overline{t_{pen}}$
1*	1.00	93.9	0.89	0.78	0.14	0.01	1.85	1.00	99.0	0.42	0.39	0.04	0.00	0.86
2	0.99	93.6	0.87	0.78	0.20	0.02	1.90	0.99	99.0	0.36	0.36	0.05	0.01	0.78
4	0.98	93.7	0.82	0.74	0.22	0.02	1.83	0.98	99.0	0.34	0.35	0.05	0.00	0.75
8	0.97	93.2	0.76	0.68	0.28	0.02	1.77	0.98	98.9	0.33	0.34	0.06	0.00	0.74
16	0.93	93.3	0.63	0.57	0.26	0.02	1.54	0.98	99.0	0.32	0.33	0.06	0.01	0.73
32	0.93	93.5	0.61	0.55	0.30	0.02	1.52	0.98	98.9	0.31	0.33	0.06	0.01	0.71
RRK								LFK15						
1*	1.00	61.6	10.07	9.41	1.67	0.29	21.66	1.00	90.3	0.95	0.79	0.22	0.02	2.01
2	1.02	53.8	10.34	9.43	2.34	0.28	22.65	1.00	89.9	0.95	0.80	0.31	0.02	2.12
4	0.99	53.7	9.91	9.01	2.68	0.26	22.13	0.97	90.2	0.84	0.70	0.35	0.02	1.96
8	0.93	55.2	9.19	8.39	2.90	0.28	21.05	0.95	90.1	0.73	0.62	0.38	0.03	1.80
16	0.81	60.6	7.63	7.28	2.83	0.30	18.30	0.93	89.7	0.61	0.54	0.42	0.03	1.66
32	0.70	63.2	6.29	6.16	2.59	0.29	15.58	0.90	89.9	0.45	0.39	0.43	0.03	1.34
BTRIX								LFK14						
1*	1.00	97.0	1.20	0.88	0.17	0.04	2.32	1.00	99.4	0.57	0.68	0.03	0.00	1.28
2	0.90	97.1	0.92	0.82	0.17	0.04	1.98	0.98	99.4	0.39	0.52	0.03	0.00	0.94
4	0.91	97.3	0.90	0.81	0.18	0.04	1.95	0.98	99.4	0.40	0.55	0.03	0.00	0.98
8	0.90	97.4	0.87	0.80	0.19	0.04	1.92	0.98	99.5	0.36	0.47	0.03	0.00	0.86
16	0.90	97.6	0.86	0.80	0.20	0.04	1.93	0.98	99.5	0.39	0.53	0.03	0.00	0.95
32	0.90	97.7	0.86	0.81	0.22	0.04	1.95	0.98	99.5	0.36	0.50	0.03	0.00	0.89
UXF								LFK8						
1*	1.00	99.1	0.15	0.14	0.04	0.00	0.34	1.00	81.7	2.62	2.41	0.48	0.07	5.67
2	1.00	99.2	0.11	0.14	0.04	0.00	0.30	0.98	81.6	2.49	2.26	0.59	0.07	5.49
4	1.00	99.1	0.10	0.12	0.05	0.01	0.28	0.96	81.5	2.43	2.20	0.70	0.06	5.49
8	1.00	99.2	0.10	0.13	0.05	0.01	0.30	0.93	81.7	2.23	2.03	0.78	0.07	5.22
16	1.00	99.1	0.09	0.13	0.07	0.01	0.30	0.83	83.7	1.72	1.59	0.75	0.07	4.23
32	1.00	99.1	0.05	0.08	0.06	0.01	0.20	0.75	85.7	1.25	1.15	0.66	0.07	3.22

* base system

Table 4.2: Increasing cache queue depths

For these two routines, queue depths greater than 8, provide the most significant improvements.

4.3. Cache banks

The previous experiments showed that as the processor queue depth increased, *proc stall* tended to decrease but *switch wait* increased. The result is that the overall system performance does not substantially increase. The *switch wait* increase is due to the increased arrival rate of the references at the switch, resulting in more conflicts. The number of cache banks can be increased to reduce the probability of conflicts in accessing the cache. This increase in the number of cache banks increases the degree of parallelism in routing references to cache banks.

In these experiments we vary the number of cache banks in the base system from 1 to 16. While the number of banks is changed, we maintain the physical capacity of the cache at our chosen size of 64Kbytes. For example, in a 2 bank system, each bank has a capacity of 32Kbytes and for a 16 bank system, there are 4 Kbytes per bank. The results are presented in Table 4.3.

The number of cache banks generally has more effect as well as a more uncertain effect on the system performance than the processor and cache queue depths. The *cp tim* shows the best improvements over the 1 to 4 bank range, with diminishing results for 8 and 16 banks. The fact that one cache bank, shared by 8 processors results in substantially worse performance than four

NEWRZ								LFK18						
cache bank	norm. cp tim	hit ratio	$\overline{t_{pstall}}$	$\overline{t_{swait}}$	miss stall	mem wait	$\overline{t_{pen}}$	norm. cp tim	hit ratio	$\overline{t_{pstall}}$	$\overline{t_{swait}}$	miss stall	mem wait	$\overline{t_{pen}}$
1	2.26	97.8	6.04	6.36	0.05	0.01	12.46	2.37	99.6	6.25	6.61	0.01	0.00	12.87
2	1.33	95.6	2.24	2.24	0.11	0.01	4.61	1.28	99.3	1.67	1.89	0.02	0.00	3.59
4*	1.00	93.9	0.89	0.78	0.14	0.01	1.85	1.00	99.0	0.42	0.39	0.04	0.00	0.86
8	0.91	92.8	0.52	0.38	0.26	0.03	1.24	0.96	98.9	0.23	0.17	0.08	0.01	0.50
16	0.89	92.5	0.45	0.31	0.47	0.04	1.35	0.94	98.8	0.16	0.09	0.14	0.01	0.43
RRK								LFK15						
1	1.38	75.3	15.93	15.04	0.88	0.22	32.19	1.79	96.3	5.04	5.20	0.10	0.01	10.36
2	1.21	65.3	13.49	12.20	1.49	0.26	27.62	1.22	93.1	2.06	1.98	0.20	0.01	4.27
4*	1.00	61.6	10.07	9.41	1.67	0.24	21.66	1.00	90.3	0.95	0.79	0.22	0.02	2.01
8	1.01	42.4	10.16	8.61	3.81	0.35	23.39	0.92	89.1	0.54	0.39	0.32	0.03	1.36
16	1.11	29.3	11.36	8.22	9.38	0.67	30.72	0.90	88.7	0.47	0.32	0.53	0.06	1.50
BTRIX								LFK14						
1	1.62	97.3	3.44	3.47	0.08	0.02	7.01	1.87	99.8	6.23	6.75	0.01	0.00	12.99
2	1.18	97.1	1.89	1.72	0.13	0.02	3.77	1.10	99.6	1.63	2.18	0.02	0.00	3.83
4*	1.00	97.0	1.20	0.88	0.17	0.04	2.32	1.00	99.4	0.57	0.68	0.03	0.00	1.28
8	0.83	97.3	0.69	0.50	0.28	0.06	1.57	0.97	99.5	0.18	0.17	0.03	0.00	0.39
16	0.87	97.6	0.90	0.59	0.49	0.11	2.16	0.96	99.5	0.13	0.09	0.06	0.00	0.29
UXF								LFK8						
1	1.02	99.5	0.76	0.96	0.02	0.00	1.74	1.96	91.5	8.29	8.40	0.21	0.04	16.97
2	1.00	99.3	0.28	0.36	0.03	0.00	0.68	1.37	85.2	4.85	4.64	0.44	0.06	10.05
4*	1.00	99.1	0.15	0.14	0.04	0.00	0.34	1.00	81.7	2.62	2.41	0.48	0.07	5.67
8	1.00	98.9	0.08	0.07	0.06	0.01	0.23	0.90	77.6	2.06	1.61	0.86	0.09	4.79
16	1.00	99.0	0.07	0.04	0.09	0.01	0.23	0.90	75.4	2.04	1.29	1.90	0.16	5.73

* base system

Table 4.3: Varying the number of cache banks

banks should be no surprise according to the early discussions since the bandwidth of one bank is totally inadequate to support eight processors at their natural request rate. Performance gains from increasing the cache banks from 4 to 16 are ≤ 13% for these eight routines, and for RRK actually degrades by 11%.

As the number of banks is increased, there is a very dramatic decrease in the *switch wait*. The decrease in *switch wait* in turn reduces the probability of a reference stall due to a full processor queue. Hence, there is an associated decrease in *proc stall*. However, increasing the number of banks also produces a negative effect, namely a lower hit ratio which increases miss traffic, thereby limiting performance gains and for RRK actually degrading the performance as the number of banks are increased to 8 or 16. These results suggest that the number of banks should not be arbitrarily increased due to two related factors; the decrease in the hit ratio and the increase in the bus traffic.

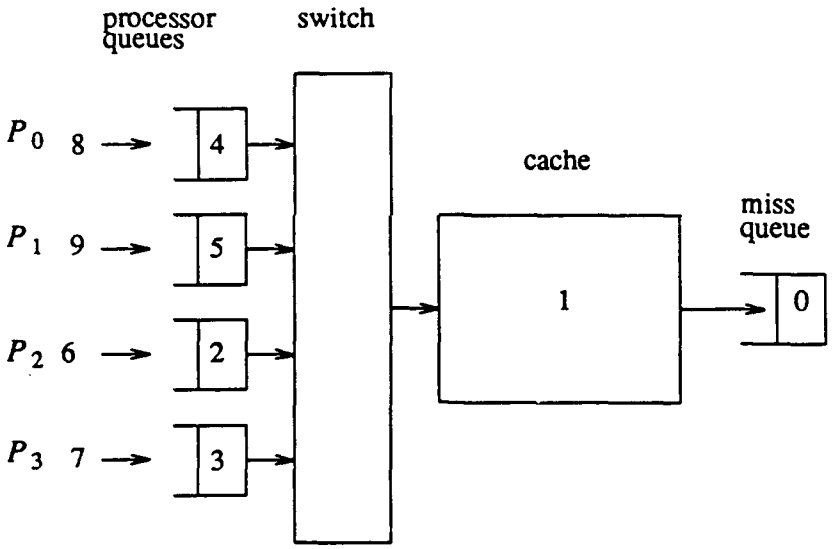
As previously explained, each cache bank has enough hardware to support an independent reference stream. As the number of banks is increased, so is the maximum buffering for miss references. In the base system, each bank has a miss queue depth of 2. A system with 8 banks has a maximum buffering of (8*2) = 16 miss queue entries. This increase in miss buffering also contributes to the lowering of the hit ratio. This phenomenon is discussed in the next section on miss queues.

Another contribution to the decreasing hit ratio is the increase in parallelism and corresponding decrease in *proc stall* and *switch wait*. Consider a system with 4 processors, P_0, P_1, P_2 and P_3, and a single cache bank with processor and miss queue depths of one, and no cache queues. Assume that each reference is to a word and that there are 4 words to a block. A particular processer, P_i, references a vector of an arbitrary length, with a base address of i and a

stride of 4 (i.e. P_i references word $i, i+4, i+8, ...$). Assume that the processors start in lock step and that the cache is initially empty. Figure 4.1a shows the system after two misses. A reference to word 0 is pending in the miss queue, waiting for memory data. Reference 1 is stalled in the cache segment because it is also a miss. It cannot be pipelined forward due to reference 0, which fills the miss queue. Reference 2 is waiting in the P_2 processor queue due to the bank-busy conflict. References 3, 4 and 5 are likewise in the respective processor processor queues. References to words 6, 7, 8 and 9 will stall the corresponding processors from issuing further references. When block 0 returns, reference 1, although it is a cache miss, will be satisfied by the block. In the following two cycles, references 2 and 3 get the cache and will hit since they also refer to block 0. Each group of four consecutive references that follow will have the same pattern of 2 misses and 2 hits.

Now consider using 4 cache banks in an effort to reduce conflicts. Figure 4.1b shows the situation when a read miss is pending in each bank. With 4 cache banks, there is a maximum miss buffering of 4. References to words 0,1,2 and 3 are to different banks and can be routed, through the switch, to the cache banks in parallel. Unlike the single bank scheme there is no conflict and hence no *proc stall* or *switch wait* for these 4 references. As they are to the same block, they are all counted as cache misses even though they will be combined. In the following cycles the processors remain in lock step and the words of each block are referenced in parallel and will also all miss. The result is that, in the single bank scheme, 50% of the references will be hits, while with 4 banks, a 0% hit ratio results. Furthermore, this decrease in the hit ratio can be attributed to the decrease in stalls and conflicts.

Increasing the number of banks results in an increase in the number of requesters on the bus. We therefore see a dramatic increase in *mem wait* for these experiments. Since *proc stall*

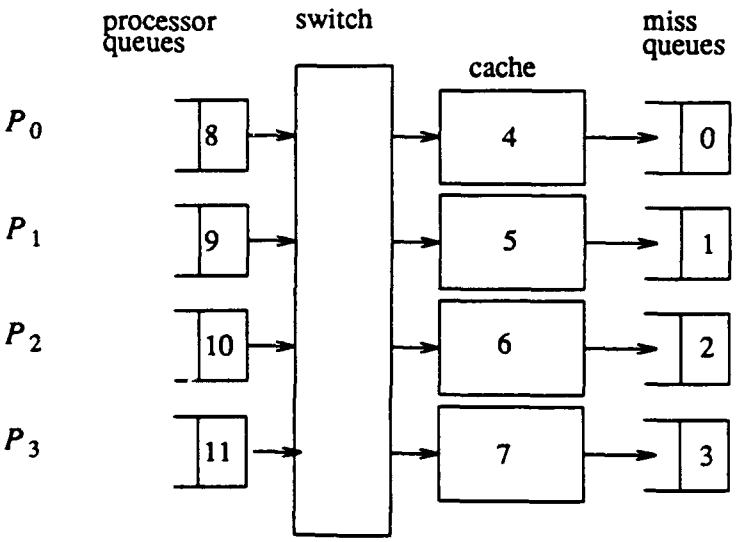


a) Single cache bank

0,1 miss in the cache, but 2 and 3 are hits due to stall

4,5 miss in the cache, but 6 and 7 are hits due to stall

etc



b) 4 cache banks

all references are cache misses

Figure 4.1: Decreasing hit ratio with an increase in the number of cache banks

and *switch wait* decrease, the time between misses must also decrease. The effect on the system from increased bus requesters is compounded by the decrease in the hit ratio. The result is an increase in *miss stall* despite the increase in maximum buffering achieved by increasing the number of cache banks.

The effect on the system of increasing the number of cache banks is most dramatic in the low hit ratio routines. Decreases in the hit ratio are relatively small for cases with ratios originally greater than 99%, but become very significant for routines with lower hit ratios. The cache hit ratio for a set of references is due to two events: first, the number of times that cache data is re-used after initial loading and secondly, the probability of using data prefetched in a block accessed from memory. The sequence in Figure 4.1 shows a reduction in the effectiveness of block access by increasing parallelism. A set of vector references that has a low hit ratio is likely to have a low data re-use. If a data set has low data re-use, the cache misses in the initial accesses to this data set has more of an effect on the overall cache hit ratio than for a data set with high data re-use. Conversely a data set with low data re-use will benefit the most from block fetching that successfully prefetches its data set before further misses to that data set occurs. When the number of cache banks is increased, the number of references that can be overlapped and queued, while they are pending to memory, also increases, and this reduces the effectiveness of block prefetching. Routines with an already low hit ratio are more significantly affected by this increase in the number of overlapped cache misses, as shown in Figure 4.1.

The performance of RRK demonstrates the failure to increase performance by increasing the number of banks. This routine has a very low hit ratio, even for a single bank. There are performance increases for 2 and 4 banks but there is also a dramatic and continuing drop in the hit ratio. When the number of banks is increased to 8, the hit ratio is so low that the performance of

the cache pipeline dominates. The increase in parallelism, compounded by the increase in the number of misses, increases *miss stall* and *mem wait*. As the miss queues become saturated, the bank-busy conflicts increase, resulting in *proc stall* increase. As the number of banks is further increased, the performance starts to fall off more significantly.

4.4. Miss reference buffering

The previous experiments indicated that the performance of the cache pipeline degrades as the number of banks is increased. To improve the performance of the cache pipeline, we look at the effect of increasing the depth of the miss queue from 1 to 32 entries. We assume that each of the four cache banks has a separate queue and that miss reference combining occurs across all banks. The results are shown in Table 4.4.

The results indicate that the best performance increase occurs over the depth range of 1 to 4 entries. As expected, the effect is more substantial for routines with a lower hit ratio, LFK8 and RRK. All the routines show substantial decrease in *miss stall*. Unlike the processor queue, which had to buffer references from switch conflicts, the range of miss queue depths is sufficient to counteract the shared bus contentions. However, it should be noted that the issue rate into the processor queue is significantly higher than the cache issue rate into the miss queues. In the processor pipeline, vector references are being made, while the cache only issues miss references to the miss queue. Furthermore, due to request combining, the number of requests that use the bus is less than the total number of misses. The decreases in *miss stall* cause decreases in *switch wait* and *proc stall*. Again routines with a low hit ratio are affected more significantly in *norm. cp tim* because the cache pipeline plays a more dominant role in their performance. The increase in miss queue depth increases *mem wait* due to increased arrival rates at the miss queues and hence increased conflicts at the bus. This increase, however, is much less than the increase due

NEWRZ								LFK18						
miss queue	norm. cp tim	hit ratio	$\overline{t_{pstall}}$	$\overline{t_{swait}}$	miss stall	mem wait	$\overline{t_{pen}}$	norm. cp tim	hit ratio	$\overline{t_{pstall}}$	$\overline{t_{swait}}$	miss stall	mem wait	$\overline{t_{pen}}$
1	1.09	95.6	1.26	1.10	0.29	0.01	2.66	1.01	99.3	0.48	0.44	0.06	0.00	0.99
2*	1.00	93.9	0.89	0.78	0.14	0.01	1.85	1.00	99.0	0.42	0.39	0.04	0.01	0.86
4	0.94	92.7	0.65	0.44	0.09	0.02	1.25	0.99	98.8	0.39	0.37	0.03	0.01	0.81
8	0.94	92.2	0.65	0.57	0.10	0.03	1.42	0.99	98.7	0.37	0.35	0.02	0.01	0.76
16	0.92	92.2	0.56	0.55	0.08	0.03	1.28	0.98	98.6	0.36	0.34	0.02	0.01	0.75
32	0.90	91.0	0.51	0.47	0.06	0.03	1.15	0.98	98.4	0.34	0.33	0.02	0.01	0.71
RRK								LFK15						
1	1.31	71.5	14.03	12.93	2.63	0.13	29.81	1.12	93.1	1.57	1.30	0.48	0.01	3.37
2*	1.00	61.6	10.07	9.41	1.67	0.29	21.66	1.00	90.3	0.95	0.79	0.22	0.02	2.01
4	1.02	41.1	10.33	9.48	2.11	0.31	22.66	0.94	89.7	0.70	0.41	0.13	0.03	1.33
8	1.04	28.6	10.55	9.57	2.07	0.36	23.13	0.92	89.2	0.57	0.59	0.08	0.03	1.34
16	1.03	25.7	10.47	9.44	1.69	0.36	22.57	0.91	89.1	0.53	0.47	0.07	0.04	1.19
32	1.02	24.2	10.44	9.47	1.49	0.36	22.41	0.90	89.4	0.50	0.44	0.06	0.03	1.10
BTRIX								LFK14						
1	1.05	97.0	1.42	1.04	0.25	0.02	2.75	1.00	99.6	0.60	0.69	0.04	0.00	1.33
2*	1.00	97.0	1.20	0.88	0.17	0.04	2.32	1.00	99.4	0.57	0.68	0.03	0.00	1.28
4	1.00	97.1	1.20	0.89	0.15	0.04	2.31	1.00	99.4	0.58	0.70	0.02	0.00	1.31
8	1.00	97.3	1.21	0.93	0.12	0.05	2.34	1.00	99.4	0.59	0.72	0.02	0.00	1.34
16	1.00	97.3	1.19	0.93	0.10	0.05	2.30	1.00	99.4	0.54	0.64	0.02	0.00	1.21
32	0.98	97.3	1.10	0.84	0.08	0.05	2.11	1.00	99.3	0.55	0.68	0.02	0.00	1.26
UXF								LFK8						
1	1.00	99.3	0.14	0.14	0.06	0.00	0.35	1.23	86.5	3.87	3.53	0.95	0.03	8.41
2*	1.00	99.1	0.15	0.14	0.04	0.00	0.34	1.00	81.7	2.62	2.41	0.48	0.07	5.67
4	1.00	99.0	0.12	0.11	0.04	0.01	0.28	0.93	74.8	2.25	2.03	0.42	0.08	4.94
8	1.00	98.9	0.12	0.13	0.02	0.01	0.29	0.83	75.9	1.70	1.48	0.31	0.09	3.77
16	1.00	98.7	0.11	0.13	0.02	0.01	0.29	0.76	77.6	1.31	1.13	0.21	0.10	2.93
32	1.00	98.7	0.09	0.12	0.01	0.01	0.24	0.64	82.8	0.68	0.59	0.07	0.10	1.58

* base system

Table 4.4: Increasing miss queue depths

increasing the number of cache banks. Furthermore, increasing the miss queue depth decreases *miss stall* whereas *miss stall* increases when the number of cache banks increases.

The purpose of pipelining miss references to memory is to reduce the effect of cache misses on other references. By increasing the depth of the miss queue, we are essentially increasing the degree of overlap for handling miss references. As with increasing the number of cache banks, the results indicate that increasing the miss queue size also generally decreases the cache hit ratio. Consider again the single cache bank system of Figure 4.1 which has a single entry miss queue. If we now increase the miss queue depth to 2, as in Figure 4.2, reference 1 can be pipelined to the miss buffer. This enables reference 2 to get the cache where it will miss and stall

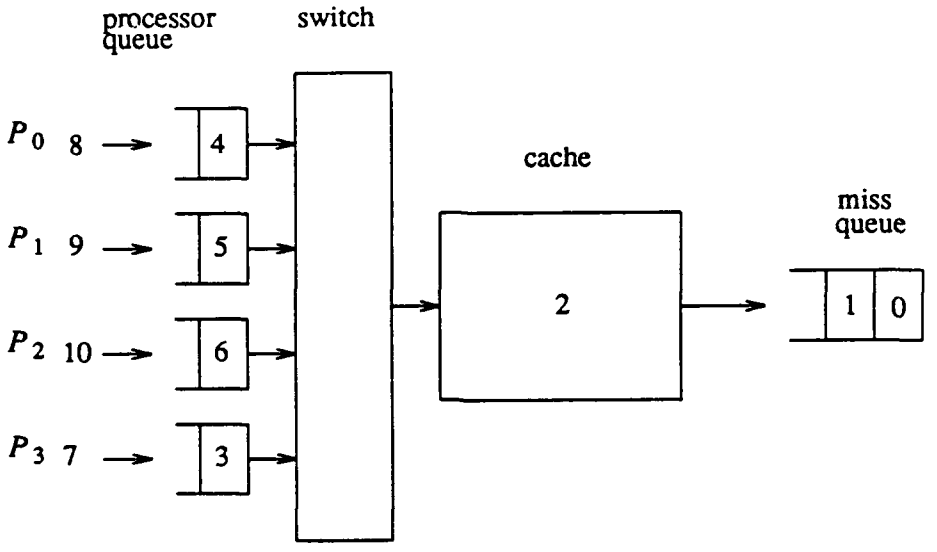


Figure 4.2: Increasing miss queues

until a queue entry is available. Reference 3 waits in the processor queue of P_3 due to the bank-busy conflict. When block 0 returns, only reference 3 will hit in the cache. The result is a decrease in the hit ratio from 0.50 to 0.25 because of the increase in overlap in processing cache misses. However, unlike increasing the number of cache banks, there is no increase in throughput in this example. Note that increasing the number of cache banks also increases the maximum buffering available, so that its effect of decreasing the hit ratio can also be partially attributed to increased buffering.

All routines show monotonic performance improvements for increases in the miss queue depth, except for RRK which has a very low hit ratio. RRK shows a significant performance improvement for increasing the queue depth from 1 to 2 entries, but with deeper queues the

dramatic increase in the number of misses exceeds the benefits initially provided by the queue as is evident by the increases in *proc stall* and *switch wait*. This result for RRK indicates that further decreases of an already low hit ratio will prevent the increased miss queue depths from achieving increases in performance.

4.5. Interleaving

Word interleaving allocates words with consecutive addresses in consecutive cache banks and memory modules. How words are allocated affects the degree of conflicts and pipeline stalls. In this set of experiments we vary the interleaving scheme of the cache banks and memory modules with respect to the base system. The results are shown in Table 4.5.

We have chosen two routines with relatively low hit rates and two with high hit rates for this study. In three of the routines (NEWRZ, LFK8 and LFK18), the base system (W/W) performs much better than block and split interleaving. In LFK 14, it performs worst. Both

	NEWRZ							LFK18						
inter-leave	norm. cp tim	hit ratio	$\overline{t_{pstall}}$	$\overline{t_{swait}}$	miss stall	mem wait	$\overline{t_{pen}}$	norm. cp tim	hit ratio	$\overline{t_{pstall}}$	$\overline{t_{swait}}$	miss stall	mem wait	$\overline{t_{pen}}$
W/W*	1.00	93.9	0.89	0.78	0.14	0.01	1.85	1.00	99.0	0.42	0.39	0.04	0.00	0.86
S/W	1.10	95.5	1.15	0.98	0.19	0.02	2.36	1.06	99.2	0.70	0.66	0.05	0.01	1.42
S/S	1.21	97.4	1.59	1.35	0.15	0.01	3.11	1.06	99.2	0.71	0.68	0.05	0.00	1.44
B/W	1.15	95.4	1.31	1.10	0.24	0.01	2.69	1.12	99.5	0.96	0.90	0.04	0.00	1.90
B/B	1.40	97.4	2.32	2.04	0.23	0.01	4.60	1.13	99.5	1.02	0.00	0.05	0.00	1.07
	LFK8							LFK14						
W/W*	1.00	81.7	2.62	2.41	0.48	0.07	5.67	1.00	99.4	0.57	0.68	0.03	0.00	1.28
S/W	1.08	86.1	3.19	2.70	0.68	0.01	6.59	0.98	99.7	0.32	0.41	0.02	0.00	0.75
S/S	1.07	91.4	3.11	2.69	0.49	0.05	6.36	0.98	99.8	0.33	0.42	0.02	0.00	0.77
B/W	1.12	86.4	3.45	2.90	0.75	0.05	7.22	0.98	99.7	0.38	0.51	0.02	0.00	0.92
B/B	1.35	90.9	4.69	4.02	0.80	0.03	9.57	0.98	99.8	0.33	0.42	0.02	0.00	0.78

* base system; W = word, B = block, S = split; a/b a = cache interleaving, b = memory interleaving

Table 4.5: Interleaving schemes

NEWRZ and LFK8 have vector accesses with strides of one and perform worst when blocks are located within one or two banks. The performance degradation of other interleave schemes relative to the base system is less for LFK18 than NEWRZ and LFK8, but this is due to the high hit ratio of LFK18, which reduces the number of multi-access conflicts when using block or split interleaving. LFK14 is the only routine that performs better for an interleave scheme other than word level. This routine uses a stride of 8 with short vector lengths, which is better suited for the block and split schemes. This can be seen from the decreases in *proc stall* and *switch wait* compared to the base system as opposed to increases for the other routines. The overall results indicate that word level interleaving at the cache and memory provides the best general purpose configuration. However, interleaving performance is very much dependent on the access pattern of a benchmark. Although we have shown only a few benchmarks here, the apparent performance for word interleaving is consistent with the results from other memory studies, and with common design practice.

The use of block level interleaving in the main memory with heavy miss traffic generally reduces *mem wait*. Recall that the various interleave schemes trade concurrency off against memory latency. Word level interleaving results in multiple banks being busy during a memory access, but all banks are being accessed in parallel. With block level interleaving only a single memory module is busy per request. The latency for block interleaving is greater than for word level interleaving but other modules are available to serve other misses, hence the decrease in *mem wait* for block interleaving when the miss traffic is sufficiently high. However, the overall completion time shows better performance for word level interleaving because the longer memory access latency for the block causes increases in *miss stall* and this is more significant than the decrease in *mem wait*. The results for these routines indicate that reducing memory latency through word interleaving is generally more effective than providing potential

parallelism through block interleaving.

For all routines there is a higher hit ratio for block and split interleaving than for word interleaving at the cache, but there is a decrease in performance. Furthermore, all the routines (except LFK14) show increases in *proc stall* relative to the base system. This increase again confirms the phenomenon that reducing *proc stall*, and thereby increasing the throughput of the cache and hence system performance, will typically result in a lowering of the hit ratio. The various examples of this phenomenon constitute the best argument for why the penalty cycles, and *proc stall* in particular, are better indicators of performance than hit ratios.

4.6. Collective changes

The previous experiments provided an assessment of the effect of varying parameter values of individual components in the system with respect to the base organization. In this section results are presented for experiments in which queues, cache banks and memory modules are changed in concert.

Each configuration is identified by 5 parameters, (*pq,cq,cb,mq,mm*), where *pq* is the processor queue depth, *cq* the cache queue depth, *cb* the number of cache banks, *mq* the miss queue depth, and *mm* the number of memory modules. The base system is therefore (1,1,4,2,4).

In this section, the results from previous experiments where only a single component of the system system was changed relative to the base system, are often used in comparisons with the results from the collective change experiments. To refer to these "single component change" configurations we use the following notation. The base system is B_0. Increasing the number of cache banks of the base system from 4 to 8, as was done in the cache bank experiments, results in a B_{8cb} or (1,1,8,2,4) configuration. Similarly increasing the cache miss queue depth, of the base system, from 1 to 4 results in a B_{4mq} or (1,1,4,4,4) configuration. The results are shown in

Table 4.6b.

4.6.1. Queues and cache banks, configuration (1,8,4,4,4) and (1,16,4,4,4)

Though implementation details have not been discussed, it is clear that the cost of increasing the number of cache banks is much higher than that of increasing the queue depths, particularly the processor and cache queues. When a similar performance is obtained by two systems that differ in the number of cache banks and processor and cache queue depths, the system with

	NEWRZ							LFK18						
system config**	norm. cp tim	hit ratio	$\overline{t_{pstall}}$	$\overline{t_{swait}}$	miss stall	mem wait	$\overline{t_{pen}}$	norm. cp tim	hit ratio	$\overline{t_{pstall}}$	$\overline{t_{swait}}$	miss stall	mem wait	$\overline{t_{pen}}$
(1,1,4,2,4)*	1.00	93.9	0.89	0.78	0.14	0.02	1.86	1.00	99.0	0.42	0.39	0.04	0.00	0.86
(1,8,4,4,4)	0.96	92.4	0.57	0.68	0.18	0.04	1.55	0.98	98.8	0.33	0.34	0.05	0.01	0.74
(1,16,4,4,4)	0.96	92.4	0.54	0.50	0.20	0.03	1.35	0.97	98.8	0.30	0.34	0.05	0.01	0.71
(1,1,8,4,4)	0.95	92.5	0.50	0.37	0.23	0.04	1.23	0.96	98.8	0.23	0.17	0.07	0.01	0.49
(1,1,8,4,8)	0.90	92.5	0.32	0.23	0.14	0.06	0.88	0.94	98.7	0.18	0.13	0.05	0.01	0.40
(1,1,8,2,16)	0.92	92.8	0.41	0.31	0.21	0.04	1.05	0.94	98.8	0.18	0.14	0.05	0.01	0.40
(1,16,16,16,16)	0.85	89.6	0.11	0.08	0.32	0.16	1.20	0.91	98.6	0.05	0.00	0.06	0.02	0.18
	LFK8							LFK14						
(1,1,4,2,4)*	1.00	81.7	2.62	2.41	0.48	0.13	5.80	1.00	99.4	0.57	0.68	0.03	0.00	1.28
(1,8,4,4,4)	0.82	79.3	1.63	1.48	0.55	0.15	4.01	0.98	99.4	0.39	0.52	0.02	0.00	0.94
(1,16,4,4,4)	0.79	79.1	1.50	1.39	0.63	0.12	3.86	0.98	99.4	0.40	0.54	0.02	0.00	0.97
(1,1,8,4,4)	0.90	76.4	2.06	1.63	0.80	0.15	4.96	0.96	99.4	0.15	0.14	0.03	0.00	0.33
(1,1,8,4,8)	0.74	77.2	1.20	0.97	0.47	0.21	3.29	0.96	99.5	0.14	0.13	0.02	0.01	0.31
(1,1,8,2,16)	0.81	76.9	1.58	1.25	0.66	0.21	4.13	0.96	99.5	0.15	0.14	0.02	0.00	0.33
(1,16,16,16,16)	0.54	83.8	0.08	0.00	0.07	0.51	1.39	0.95	99.3	0.05	0.05	0.03	0.01	0.18

* base system

Table 4.6a: Collective changes

	NEWRZ							LFK18						
system config**	norm. cp tim	hit ratio	$\overline{t_{pstall}}$	$\overline{t_{swait}}$	miss stall	mem wait	$\overline{t_{pen}}$	norm. cp tim	hit ratio	$\overline{t_{pstall}}$	$\overline{t_{swait}}$	miss stall	mem wait	$\overline{t_{pen}}$
(1,8,4,2,4)	0.97	93.2	0.76	0.68	0.28	0.02	1.77	0.98	98.9	0.33	0.34	0.06	0.00	0.74
(1,16,4,2,4)	0.93	93.3	0.63	0.57	0.28	0.02	1.54	0.98	99.0	0.32	0.33	0.06	0.01	0.73
(1,1,8,2,4)	0.91	92.8	0.52	0.38	0.26	0.03	1.24	0.96	98.9	0.23	0.17	0.08	0.01	0.50
(1,1,16,2,4)	0.89	92.5	0.45	0.31	0.47	0.04	1.35	0.94	98.8	0.16	0.09	0.14	0.01	0.43
(1,1,4,4,4)	0.94	92.7	0.65	0.44	0.09	0.02	1.25	0.99	98.8	0.39	0.37	0.03	0.01	0.81
	LFK8							LFK14						
(1,8,4,2,4)	0.93	81.7	2.23	2.03	0.78	0.07	5.22	0.98	99.5	0.36	0.47	0.03	0.00	0.86
(1,16,4,2,4)	0.83	83.7	1.72	1.59	0.75	0.07	4.23	0.98	99.5	0.39	0.53	0.03	0.00	0.95
(1,1,8,2,4)	0.90	77.6	2.06	1.61	0.86	0.09	4.79	0.97	99.5	0.18	0.17	0.03	0.00	0.39
(1,1,16,2,4)	0.90	75.4	2.04	1.29	1.90	0.16	5.73	0.96	99.5	0.13	0.09	0.06	0.00	0.29
(1,1,4,4,4)	0.93	74.8	2.25	2.03	0.42	0.08	4.94	1.00	99.4	0.58	0.70	0.02	0.00	1.31

** (pq,cq,cb,mq,mm)

Table 4.6b: Previous results referred to in section 4.6

fewer cache banks would generally be preferred. The previous experiments suggest that increasing the cache queue depth reduces the *proc stall* and *switch wait*, but increases the *miss stall*. Increasing the miss queue depth reduces the *switch wait* and the *miss stall*. Using a system with 4 cache banks, we consider what processor and cache queue depths would result in an overall performance (*cp time*) comparable to a base system with 8 cache banks i.e. B_{8cb}. In the first configuration the cache queue depth is increased to 8 and the miss queue depth to 4 with respect to the base system, i.e. (1,8,4,4,4). In the second experiment the cache queue depth is further increased from 8 to 16, for a (1,16,4,4,4) configuration.

As with previous results, increasing the cache queue reduces *proc stall* and *switch wait* of (1,8,4,4,4) relative to B_0. All routines show overall performance increases (decreases in *cp time*) with respect to B_0, but the overall system performance is similar to a base system with 8 cache queues i.e. B_{8cq}, except for LFK8 where the performance of (1,8,4,4,4) is much better than B_{8cq}. LFK8 has a significantly lower hit ratio than the other routines. The two additional miss queue positions in (1,8,4,4,4) with respect to B_{8cq} helps to reduce the *miss stall* of LFK8 from 0.78 for B_{8cq} to 0.55 for (1,8,4,4,4). However a reduction in *miss stall* does not necessarily indicate improved performance, for instance B_0 shows the lowest values for *miss stall* of the three configurations (B_0, B_{8cq} and (1,8,4,4,4)), but this is because the short cache queue in B_0 results in more bank busy and switch conflicts (higher values for *proc stall* and *switch wait*) which reduces rate of references accessing the cache and consequently the rate of misses entering the miss queues.

In comparing the overall system performance of (1,8,4,4,4) with a base system with miss queues of depth 4, i.e. B_{4mq}, the overall performance of NEWRZ is slightly worse for (1,8,4,4,4) than for B_{4mq}, but LFK18 and LFK14 are slightly better with (1,8,4,4,4). While deep cache

queues help the processor pipeline, the effect of increasing the miss reference arrival rate is to create additional stalls at the miss queue. For instance in NEWRZ, B_{4mq} has a *miss stall* of 0.09 cycles per reference while configuration (1,8,4,4,4) has a *miss stall* of 0.18 cycles per reference.

LFK8 again shows a much higher overall performance with (1,8,4,4,4) than with B_{4mq}. With (1,8,4,4,4), LFK8 has a higher value of *miss stall* than for the configurations B_0 and 8_{4mq}, but has a lower value for *proc stall* and *switch wait.*

Increasing the cache queue depth of configuration (1,8,4,4,4) to 16 i.e. (1,16,4,4,4), result in some reductions in *proc stall* and *switch wait*, but only LFK8 and LFK18 show even a minimal improvement in *cp time*.

4.6.2. Configurations (1,1,8,4,4), (1,1,8,4,8), and (1,1,8,2,16)

In the next configuration the number of cache banks is increased to 8. The cache bank experiments showed that increasing the number of cache banks increased the rate of references passing through the cache segment including cache misses. To handle this increase in the arrival rate of the misses, the miss queue depth is increased to 4. As the effect of the cache queues on the processor pipeline is small compared with the cache bank increase, we keep its depth at the minimum. This configuration is (1,1,8,4,4).

As expected the system performs better (lower *cp time*) than the previous "multiple change" and B_0 configurations except for LFK8. LFK8 has a low hit ratio and the increase in the number of cache banks and cache queues depth lowers the hit ratio, increasing the number of misses handled in the cache. For all routines the (1,1,8,4,4) configuration increases the *miss stall* over the previous configurations but only LFK8, which suffers a large *miss stall* increase, shows a decrease in overall performance as a result.

When (1,1,8,4,4) is compared to a base system with 8 cache banks, i.e. B_{8cb} (where the difference between the two configurations is only in the depth of the miss queues), three of the routines (LFK8, LFK14 and LFK18) show comparable *cp time* to B_{8cb}, but for NEWRZ, (1,1,8,4,4) shows a higher *cp time* when compared with B_{8cb}. Both LFK8 and NEWRZ have relatively low hit ratios, but the shorter miss queue of B_{8cb} results in a much better overall performance for B_{8cb} over (1,1,8,4,4). Comparing the performance of LFK8 with (1,1,8,4,4), (1,16,4,4,8) and (1,8,4,2,4) shows that increasing the number of cache banks to 8 has both increased the number of misses and increased their arrival rate. Since the memory system is unable to service the requests at a suitable rate, it now becomes the limiting factor.

Given a fixed memory module access latency, the only option available for improving the memory system's performance is to increase the number of memory modules. In the next configuration the number of memory modules is increased from 4 to 8 i.e. (1,1,8,4,8,). Table 4.6a shows that this provides a good performance improvement compared to B_0 for all routines, and has equal or better performance than (1,1,8,4,4) and much better for LFK8. The increased number of memory modules compensate for the increased number of cache banks and reduces the *miss stall* cycles to a level comparable with B_0. Furthermore, the increased performance of the miss handling by the cache generally reduces *proc stall* and *switch wait* slightly (when compared with B_{8cb}) such that the overall performance is close to that of B_{16cb} for NEWRZ, LFK18 and LFK15, and significantly exceeds it for LFK8. The performance of LFK8 with (1,1,8,4,8) exceeds the performance with (1,1,8,4,4) since increasing only the number of cache banks increases the number of misses to the miss queues resulting in a *miss stall* penalty of 0.8. When the memory modules are increased as well, more memory modules are available for servicing misses and *miss stall* for LFK8 is reduced to 0.47. This result supports the notion that the previous (1,1,8,4,4) configuration was limited by memory service for LFK8.

Surprisingly there is an increase in the *mem wait* for NEWRZ and LFK8 for both (1,1,8,4,4) and (1,1,8,4,8), relative to B_0. These two routines also showed the largest decreases in the hit ratio when the number of cache banks were increased from 4 to 8 (B_0 to B_{8cb}). Intuitively, the hit ratio may have been expected to increase again when more memory modules were added, but the increases, if any, are modest and do not approach the hit ratio of B_0. However, recall that *mem wait* is the time taken to gain access to and use the bus. *Mem wait* includes any cycles necessary to get the bus only to find the requested module busy. Returning data has the highest priority for the bus. The increase in the number of higher priority bus requesters, i.e. memory modules, increases *mem wait* for the processors. But this increase only degrades the overall performance by a small fraction.

These results indicate that increasing the number of memory modules reduces the memory as a performance bottleneck. In the next configuration the number of memory modules is increased to 16 while decreasing the number of miss queue entries to 2, i.e. (1,1,8,2,16). Just decreasing the miss queue depth would normally lead to an increase in *miss stall*. However, by increasing the number of memory modules we hope to minimize such an effect. Furthermore, reducing the miss queue depth should improve the hit ratio. Therefore, the tradeoff is cache throughput versus memory throughput.

The results, contrary to the hopes stated above, show that the overall system performance for (1,1,8,4,8) and (1,1,8,2,16) is about the same for the two routines with the high hit ratios (LFK18 and LFK14), but worse for the high miss ratio routines (NEWRZ and especially for LFK8). In the high hit ratio routines there is only a small amount of miss traffic, hence the changes in the miss queue depth and memory modules have little effect on the performance. For NEWRZ and LFK8, even though the increase in the number of memory modules provide for

potential parallelism among memory accesses, the shorter miss queue prevents the system from exploiting it effectively. The short miss queue has become the limiting factor in their performance. The respective *miss stall* has increased from 0.14 for (1,1,8,4,8) to 0.21 for (1,1,8,4,16) for NEWRZ and from 0.47 to 0.66 for LFK8.

4.6.3. Configuration (1,16,16,16,16)

In this last configuration, the system size is dramatically increased. As with most of the previous results, the behavior of the system depends on the hit ratio of the routines. For the routines with a high hit ratio the overall performance of (1,16,16,16,16) is slightly better than for (1,1,8,4,8). The combination of more cache banks and very deep cache queues virtually eliminates *proc stall* and *switch wait* and the processor pipeline performance approaches its maximum.[1] The overall performance improvement is mainly due to the processor pipeline. Further system performance improvements resulting from enhancing miss servers are unlikely to be significant as the number of misses is very low. Though *mem wait* increases, it does not contribute a significant amount to the number of penalty cycles. Hence, any attempts to reduce this factor would not yield any real performance improvements over the (1,16,16,16,16) configuration.

In the low hit ratio routines, LFK8 and NEWRZ, significant system performance improvements are seen with (1,16,16,16,16). The processor pipeline also shows dramatically less *proc stall* and *switch wait*. Compared to the configurations (1,1,8,4,8) and (1,1,8,2,16) the absolute decrease in stall and wait cycles in the processor pipeline is greater than for the high hit ratio routines, which has not been the general trend with previous configurations. The performance of

[1] The maximum performance for the processor pipeline is when *proc stall* = *switch wait* = 0.

the cache pipeline also shows some contrary trends from earlier configurations. In NEWRZ, the cache pipeline performance is worse than the previous two configurations, so the overall performance improvement is due to the processor pipeline. However although *mem wait* in LFK8 increased, *miss stall* was reduced dramatically and the result is a performance improvement substantially greater than any previous configuration.

4.7. Performance indices

In the experiments that varied the processor and cache queue depths, the results generally indicate small increases of performance with increasing queue depths beyond those of the base system. The cache miss ratio, however, remains relatively constant across changes in the queue depths. For example, the results for LFK8 with cache queue depths of 1 (the base system) and 16 both have a hit ratio of 81.7%, but the completion time and $\overline{t_{pen}}$ decrease by approximately 7% and 8.5%, respectively. As the role of the queues is simply to buffer references before the cache, a relatively constant cache hit ratio over variations in the queue depths is to be expected. The highest percentage difference in the cache hit ratio is approximately 5%. The difference can probably be attributed to some re-ordering of the references that can occur due to the buffering. The index $\overline{t_{pen}}$ does provide a general indication of performance, but it is not a linear relation and does not always even change in the same direction, as noted in the experiments with increasing processor queues. This may be due to inaccuracies in the model for $\overline{t_{pen}}$ and the simulation model. For instance, $\overline{t_{pen}}$ does not account for the time between issuing references and the simulation model does no detailed modeling of how cache and memory data is returned to the processors. However, since the performance of the memory system, miss queues and the cache ultimately affect the issuing of references by the processor pipeline, *proc stall* seems to provide a generally reliable indication of performance for this type of system.

The experiments with increasing the number of cache banks and miss queue depth show how a simple cache hit ratio (that does not account for queue matched misses) can provide a very misleading indication of system performance. In both simulation cases the routines generally show a consistent decrease in the hit ratio, but the completion time and $\overline{t_{pen}}$ indicate a performance increase. As discussed in sections 4.4 and 4.5, this miss ratio effect is due to the increase in concurrency or overlap in handling closely issued references to the same block. With the increase in overlap, references to the same block can be moved past the cache before the first miss to the block can fetch the data, whereas without overlap, multiple references to the same block are often stalled before addressing the cache until the memory data is returned and therefore becoming hits. If the highest cache hit ratio can be obtained by stalling a pipeline, this suggests that the cache hit ratio is an incorrect performance index for this type of system.

The conditions that result in a low hit ratio in the system, i.e. closely issued references to the same block and frequent cache loading, is characteristic of vectorizable program loops. For these programs a cache that handles miss references concurrently will result in poorer performance unless a mechanism such as the matching miss queue is used to combine misses to the same block. In our system, it is this mechanism that allows the performance to increase while the cache hit ratio decreases. Figure 4.7 shows the percentage of miss references that matched a pending request in the miss queue. The column *max depth* is the total number of buffer positions in the system for reference misses. Results show that as the buffering increases the percentage of the references combined also increase. However, we have only seen significant increases in performance for some of the routines as the number of cache banks and the cache miss queue depth is increased from the minimal level up to levels found in the base system. This suggests that while combining is important it is only one component needed for high performance since increasing the buffering can also create bottlenecks elsewhere in the system.

cache banks	max. depth	NEWRZ	LFK18	LFK8
1	2	30.74	24.61	28.51
2	4	50.04	41.24	48.25
4*	8	64.94	54.80	62.77
8	16	68.93	59.71	69.49
16	32	72.34	63.05	71.42

miss queue	max. depth	NEWRZ	LFK18	LFK8
1	4	32.46	24.52	29.05
2*	8	64.94	54.80	62.77
4	16	74.26	62.28	72.55
8	32	74.11	72.34	75.86
16	64	78.54	77.21	79.03
32	128	82.80	81.00	83.10

* base system

Table 4.7: Percentage of miss references that are combined in the miss queue

CHAPTER 5

CONCLUSIONS

This thesis has reported some results from simulation experiments for a moderate-size multiprocessor with a shared cache. The disadvantage of a shared cache organization is the potential for access conflicts. Approaches to reducing and minimizing the effect of these conflicts, through the use of buffer queues, pipelining, increasing the number of cache banks and interleaving, were discussed. A request matching and combining scheme was used to reduce the number of requests issued to the memory and a modified write back algorithm was used to reduce cache pipeline disruptions. Simulation experiments were conducted to examine the effect of the techniques discussed on a set of vectorizable benchmark routines.

The general performance trend, as the result of changing simulation parameters, namely processor, cache and miss queue depths, and number of cache banks, is shown in Table 5.1. Increasing the simulation parameters in all cases generally improved performance but the improvement was most significant for increasing the number of cache banks and miss the queue depth. A "-" in the table indicates a relatively constant value over the range of values used for the simulation parameter.

The following are the highlights of the results.

(1) Varying the depth of the processor and cache queues has a limited effect on the overall system performance, compared with increasing the number of cache banks and the miss queue depth. The processor and cache queue depths had more influence on routines with higher hit ratios (greater than 99%). Lower hit ratio routines tended to have constantly full queues

parameter	hit ratio	proc stall	switch wait	miss stall	mem wait
proc queues	-	decrease	increase	-	-
cache queues	-	decrease	decrease	increase	-
cache banks	decrease	decrease	decrease	increase	increase
miss queues	decrease	decrease	decrease	decrease	increase

Table 5.1: Summary of effects of increasing design parameter values

which would need much deeper queues than the largest depths used in these experiments to see an effect on system performance. The depth of the cache queue had slightly more effect than the processor queue depth. Even if the queues were made extremely deep, performance benefits could well be limited due to an inability of the processor to continue issuing requests when there are a large number of pending request already issued. This processor limitation was not modeled in our experiments.

(2) Increasing the number of cache banks or the miss queue depth reduced the number of processor queue stalls and switch conflicts. With more banks, the degree of parallel access increases and the probability of multi-access conflicts decreases. With deeper miss queues, miss references can be removed from the cache segment, making it available for new requests and thereby reducing the effect of cache misses on the processor pipeline. There are, however, two negative effects that suggest that the number of banks and queues should not be increased arbitrarily. First, as the number of cache banks and the miss queue depth are increased, the cache hit ratio is decreased. This effect is more significant for routines with an already low hit ratio (less than 90%), where it can result in performance degrading stalls in the pipelines. Secondly, increasing the throughput of the cache segment increases

the rate of arrivals into the miss queue and moves the performance bottleneck into the cache pipeline. The problem is exacerbated by the increased number of misses due to the lower hit ratio, which increases the contribution of the cache pipeline to the average number of penalty cycles per reference.

(3) To improve the performance of the system and to ensure coherence, request combining is used in the miss queues. Combining miss requests is very important for highly overlapped miss handling, particularly for vector accesses, a multi-bank shared cache, and deep miss queues. A reference can result in one of three actions, a hit in the cache, a cache miss that leads to a memory request, or a cache miss that does not generate a memory request due to its being combined with a previous miss request. The percentage of miss requests that are combined with a previous miss increases dramatically as the number of cache banks and the miss queue depth are increased.

(4) Word interleaving for both the cache banks and the memory modules yields higher performance than split or block interleaving for most benchmarks. However, it should be noted that the code used here was not re-optimized for particular variations in system configuration. It is possible that a re-distribution of the loop iterations among the processors could make block interleaving far more effective.

(5) Although the hit ratio decreases with increases in the number of cache banks and the read miss queue depth, the overall performance shows improvement. In a system that allows for high overlap of cache accesses and servicing of cache misses, the cache miss ratio is not a good indicator of overall system performance. The average number of cycles that a reference stalls and waits, $\overline{t_{pen}}$, can provide an indication of the performance trend. Furthermore since any stalls and waits ultimately affect the processor and average processor perform-

ance is very closely related to system performance, *proc stall* is perhaps the best indication of system performance. The value of *proc stall* computed in our work, however, models only stalls due to blocked requests. It does not model processor stalls due to waiting for the information in the returned data from a previous request which may be needed in order to construct further requests. Consequently, $\overline{t_{pen}}$ used in conjunction with *proc stall* gives a more complete indicator of performance.

REFERENCES

[Alli85] Alliant Computer Systems Corporation, *FX/Series Product Summary*, June 1985.

[AnST67] D. W. Anderson, F. A. Sparacio and R. M. Tomasulo, "The IBM System/360 Model 91: Machine Philosophy and Instruction Handling," *IBM Journ. of Res. and Dev.*, pp 34-53, Jan. 1967.

[ArBa84] J. Archibald and J. L. Baer, "An Economical Solution to the Cache Coherence Problem," *11th Int. Symp. on Comp. Arch.*, pp 355-562, June 1984.

[BrDa77] F. A. Briggs and E. S. Davidson, "Organization of Semi-conductor Memories for Parallel Pipelined Processors," *IEEE Trans. Comp.*, vol. C-26, pp 162-169, Feb. 1977.

[Buck78] J. K. Buckle, *The ICL2900 Series*, The Macmillan Press, 1978.

[CeFe78] L. M. Censier and A P. Feautrier, "A New Solution to Coherence Problems in Multicache Systems," *IEEE Trans. Comput.*, vol C-27, pp 1112-1118, Dec. 1978.

[ChVe88] H. Cheong and A. Veidenbaum, "A Cache Coherence Scheme with Fast Selective Invalidation," *Proc. 15th. Int. Symp. on Comp. Arch.*, pp. 299-307, June 1988.

[FuKH87] J. Fu, J. B. Keller and K. J. Haduch, "Aspects of the VAX 8800 CBox Design," *Digital Technical Journal*, pp 41-51, Feb. 1987.

[HwBr84] K. Hwang and F. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, Inc., 1984.

[Kogg81] P. M. Kogge, *The Architecture of Pipelined Computers*, McGraw-Hill, New

York, 1981.

[KDLS86] D. J. Kuck, E. S. Davidson, D. H. Lawrie and A. H. Sameh, "Parallel Supercomputing Today and the Cedar Approach," *Science*, vol. 231, pp. 967-974, February 1986.

[Lee87] R. L. Lee, "The Effectiveness of Caches and Data Prefetch Buffers in Large-scale Shared Memory Multi-processors," Center for Supercomputing Res. and Dev., Univ. of Ill., Tech Rep. 670, May. 1987.

[PaPa84] A M. Papamarcos and J. H. Patel, "A Low-overhead Coherence Solution for Multi-procesors with Private Cache Memories," *Proc. 11th. Ann. Int. Symp. on Comp. Arch.*, pp 348-354, June 1984.

[PaDa76] J. H. Patel and E. S. Davidson, "Improving the throughput of a Pipeline by insertion of delays," *Proc. of 3rd. Ann. Symp. on Computer Architecture*, pp 159-163, 1976.

[PlDa83] A. R. Pleszkun and E. S. Davidson, "A Structured Memory Access architecture," *Int. Conf. on Parallel Proc.*, pp 461-471, Aug. 1983.

[Mcma86] F. H. McMahon, "The Livermore Kernels: A Computer Test of the Numerical Performance Range," UCRL-53745, Lawrence Livermore National Laboratory, Livermore CA, Dec. 1986.

[RamLi] C. V. Ramamoorthy and H. F. Li, "Pipeline Architecture," *ACM Computing Surveys*, pp 61-102, March 1977.

[Schw86] H. Schwetman, "CSIM reference Manual(Rev. 9)," *Microelectronics and Computer Technology Corporation*, Apr. 1986.

[Smith82] A. J. Smith, "Cache memories," *ACM Computing Surveys*, vol. 14, no. 3 , pp

473-530, Sept. 1982.

[Smith84] J. E. Smith, "Decoupled Access/Execute Computer Architectures," *ACM Trans. on Computer Systems*, vol. 2, no. 4, pp 289-308, Nov. 1984.

[Smith87] J. E. Smith, et al, "The ZS-1 Central Processor," *Proc. of ASPLOS II*, pp 199-204, Oct. 1987.

[Tang76] C. K. Tang, "Cache System Design in the Tightly Coupled Multi-processor System," *AFIPS Proc. NCC*, vol. 45, pp 749-753, 1976.

[Toma76] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal of Research and Development*, vol. 11, no. 1, pp 25-33, Jan. 1976.

[Yeh81] P. C. Yeh, "Shared Cache Organization for Multiple-Stream Computer Systems," Coordinated Science Lab, Univ. of Ill., Tech Rep. R-904, Jan. 1981.